Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico Integrador

23 de Febrero de 2025

Damaris Juares
108566

Lucas Perez Esnaola
107990

Índice

1. PARTE 1 **5**
1.1. Introducción . . . 5
1.1.1. Contexto . . . 5
1.1.2. Consigna . . . 5
1.2. Análisis del problema . . . 6
1.2.1. Descripción de problema . . . 6
1.2.2. Propuesta de algoritmo greedy . . . 6
1.3. Demostración de optimalidad del algoritmo . . . 8
1.3.1. Teorema de la solución óptima . . . 8
1.3.2. Justificación . . . 8
1.4. Algoritmo propuesto . . . 9
1.4.1. Código . . . 9
1.4.2. Análisis de la complejidad . . . 9
1.4.3. Impacto de la variabilidad de valores a la complejidad temporal . . . 9
1.4.4. Impacto de la variabilidad de valores a la optimalidad . . . 9
1.5. Mediciones de tiempo . . . 11
1.5.1. Generador de sets aleatorios . . . 11
1.5.2. Mediciones . . . 11
1.6. Conclusiones . . . 13
1.7. Anexo correcciones . . . 14
1.7.1. Demostración de optimalidad del algoritmo . . . 14
1.7.2. Impacto de la variabilidad de valores a la complejidad algorítmica— . . . 16

2. PARTE 2 **17**
2.1. Introducción . . . 17
2.1.1. Contexto . . . 17
2.1.2. Consigna . . . 17
2.2. Análisis del problema . . . 18
2.2.1. Descripción de problema . . . 18
2.2.2. Propuesta de algoritmo dinamico . . . 18
2.3. Demostración de ecuación de recurrencia . . . 20
2.3.1. Planteamiento de la Ecuación de Recurrencia . . . 20
2.3.2. Demostración por Inducción . . . 20
2.3.3. Correctitud de la Ecuación . . . 21
2.3.4. Conclusión . . . 21
2.4. Algoritmo en programación dinámica . . . 22
2.4.1. Código . . . 22
2.4.2. Análisis de la complejidad . . . 22
2.4.3. Análisis del impacto de la variabilidad de los valores de las monedas en los tiempos del algoritmo . . . 22

2.5. Análisis temporal 24
2.5.1. Generador de sets aleatorios 24
2.5.2. Mediciones 24
2.6. Conclusiones 26
2.7. Anexo Correcciones 27
2.7.1. Ecuacion de recurrencia 27
2.7.2. Demostración de la Ecuacion de Recurrencia 28
2.7.3. Algoritmo en Programacion Dinámica 29
2.7.4. Mediciones 32

3. PARTE 3 **34**
3.1. Introducción 34
3.1.1. Contexto 34
3.1.2. Consigna 34
3.2. Demostración NP 36
3.2.1. Algoritmo certificador eficiente 36
3.2.2. Detalles del algoritmo 36
3.2.3. Conclusión 38
3.3. Demostración NP Completo 39
3.3.1. Problema de Bin-Packing en version unaria 39
3.3.2. Bin-Packing $\leq_P$ Batalla Naval 39
3.3.3. Conclusión 40
3.4. Backtracking 41
3.4.1. Aclaraciones previas 41
3.4.2. Detalles del algoritmo 41
3.4.3. Resultados 42
3.4.4. Ananilis de los resultados 43
3.5. Programación Lineal 43
3.5.1. Modelo planteado 44
3.5.2. Detalles del algoritmo 45
3.5.3. Resultados 46
3.5.4. Análisis de los resultados 46
3.6. Algoritmo de Aproximación 47
3.6.1. Algoritmo propuesto 47
3.6.2. Análisis de la complejidad 48
3.6.3. Mediciones 49
3.6.4. Análisis de la aproximación 49
3.6.5. Conclusión 50
3.7. Conclusiones 51
3.7.1. Características de los algoritmos 51
3.7.2. Observaciones 51
3.7.3. Conclusión Final 51

3.8. Anexo Correcciones 52
3.8.1. Mejorar la explicacion del modelo de programación lineal 52
3.8.2. Detalles del algoritmo de Backtracking 54
3.8.3. Corrección complejidad de certificador polinomial 55
3.8.4. Mejora en medición de complejidad empírica 57
3.8.5. Corrección implementación de Algoritmo de Aproximación 60

1. PARTE 1

1.1. Introducción

1.1.1. Contexto

Cuando Mateo nació, Sophia estaba muy contenta. Finalmente tendría un hermano con quien jugar. Sophi tenía 3 años cuando Mateo nació. Ya desde muy chicos, ella jugaba mucho con su hermano.

Pasaron los años, y fueron cambiando los juegos. Cuando Mateo cumplió 4 años, el padre de ambos le explicó un juego a Sophia: Se dispone una fila de n n monedas, de diferentes valores. En cada turno, un jugador debe elegir alguna moneda. Pero no puede elegir cualquiera: sólo puede elegir o bien la primera de la fila, o bien la última. Al elegirla, la remueve de la fila, y le toca luego al otro jugador, quien debe elegir otra moneda siguiendo la misma regla. Siguen agarrando monedas hasta que no quede ninguna. Quien gane será quien tenga el mayor valor acumulado (por sumatoria).

El problema es que Mateo es aún pequeño para entender cómo funciona esto, por lo que Sophia debe elegir las monedas por él. Digamos, Mateo está "jugando". Aquí surge otro problema: Sophia es muy competitiva. Será buena hermana, pero no se va a dejar ganar (consideremos que tiene 7 nada más). Todo lo contrario. En la primaria aprendió algunas cosas sobre algoritmos greedy, y lo va a aplicar.

1.1.2. Consigna

1. Hacer un análisis del problema, y proponer un algoritmo greedy que obtenga la solución óptima al problema planteado: Dados los n valores de todas las monedas, indicar qué monedas debe ir eligiendo Sophia para si misma y para Mateo, de tal forma que se asegure de ganar siempre. Considerar que Sophia siempre comienza (para sí misma).

2. Demostrar que el algoritmo planteado obtiene siempre la solución óptima (desestimando el caso de una cantidad par de monedas de mismo valor, en cuyo caso siempre sería empate más allá de la estrategia de Sophia).

3. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta la variabilidad de los valores de las diferentes monedas a los tiempos del algoritmo planteado.

4. Analizar si (y cómo) afecta la variabilidad de los valores de las diferentes monedas a la optimalidad del algoritmo planteado.

5. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también.

6. Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración. Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos. Para esto, proveemos una explicación detallada, en conjunto de ejemplos.

7. Agregar cualquier conclusión que les parezca relevante.

1.2. Análisis del problema

1.2.1. Descripción de problema

El problema para el Juego de Hermanos en su version codiciosa es usar una regla simple para seleccionar las monedas de Sofia y las de Mateo, esta regla tiene que poder llevarnos a que Sofia siempre gane las partidas acumulando una suma mayor a la de su hermano.

1.2.2. Propuesta de algoritmo greedy

Lo primero que necesitamos es elegir una norma de eleccion que cumpla con nuestro objetivo: que Sofia gane.

A continuación mostraremos el análisis de las diferentes alternativas que propusimos, y como llegamos a la alternativa elegida:

Primera propuesta Como primera idea para la regla greedy se nos ocurrió elegir la moneda de mayor valor (entre las disponibles segun las reglas del juego) en cada turno de Sofia y en el turno de Mateo utilizar el mismo criterio. Para este caso, si bien siempre empieza jugando Sofia y eso le da una ventaja, tener el mismo criterio para ambos jugadores no le garantiza a nadie la victoria. Si tomamos como ejemplo a las monedas [6,8,1,2,4,5], siguiendo nuestra propuesta de regla, las elecciones para cada jugador se verían de la siguiente manera:

Elecciones SOFIA: 6, 5, 2. Suma: 13

Elecciones MATEO: 8, 4, 1. Suma: 13

Lo cual no nos lleva a la solución buscada.

Segunda propuesta

Como segunda alternativa, propusimos elegir la moneda de mayor valor en cada turno de Sofia y en el turno de Mateo elegir la moneda rechazada por Sofia.

Elecciones SOFIA: 6, 8, 2. Suma: 16

Elecciones MATEO: 5, 4, 1. Suma: 11

En este caso nuestra eleccion cumple con nuestros requisitos. Sin embargo, no siempre devuelve la solución óptima.

En este otro caso, con monedas [6,2,1,2,4,5], las elecciones segun nuestra regla greedy serian:

Elecciones SOFIA: 6, 4, 2. Suma: 12

Elecciones MATEO: 5, 2, 1. Suma: 8

Cuando la solución óptima es:

Elecciones SOFIA: 6, 5, 4. Suma: 15

Elecciones MATEO: 2, 1, 2. Suma: 5

Propuesta final Nuestra propuesta final como forma de tomar nuestra decision sobre cada turno es mantener el criterio de seleccion anterior para Sofia pero con la modificacion de elegir la menor moneda de menor valor entre las restantes para Mateo.

Para el primer ejemplo dado, con monedas [6,8,1,2,4,5], los jugadores harían las siguientes elecciones:

Elecciones SOFIA: 6, 8, 4. Suma: 18

Elecciones MATEO: 5, 1, 2. Suma: 8

Esta regla nos lleva a nuestro objetivo de que gane Sofía, además de dar la solución óptima.

Supondremos que Sofia es muy codiciosa y elegiremos la ultima opcion descripta para asegurale la mayor suma posible ademas de ganar.

Pseudocódigo **Algoritmo Greedy:**

1. **Entrada:** Una lista de valores de las monedas.
2. **Proceso:** En cada turno:
   - Sophia elige la moneda de mayor valor entre la primera y la última de la fila.
   - Luego, Mateo elige la siguiente moneda de menor valor entre las dos primera y ultima opciones restantes.
3. **Repetir** hasta que no haya más monedas.
4. **Salida:** La lista de monedas que Sophia elige y la lista de monedas que Mateo elige, junto con la suma de las monedas elegidas por ambos.

Este enfoque asegura que Sophia siempre tomará la mejor opción posible en su turno, aprovechando el principio greedy de tomar la decisión óptima local en cada paso.

1.3. Demostración de optimalidad del algoritmo

1.3.1. Teorema de la solución óptima

El algoritmo greedy propuesto es óptimo porque, en cada turno, Sophia toma la moneda que maximiza su ganancia local. No se necesitan decisiones complejas que impliquen mirar más allá de un turno. Cada vez que toma la moneda de mayor valor, minimiza las opciones favorables para Mateo en su turno. Esto es suficiente para garantizar que Sophia obtenga el máximo valor posible.

1.3.2. Justificación

Como Sophia toma las mejores monedas para sí misma en cada turno, el algoritmo nunca deja de maximizar su ganancia, lo que lleva a una solución óptima. Por el otro lado, Mateo siempre toma las peores monedas, por lo que siempre esta minimizando su ganancia. De esta forma, siempre se llega a un resultado óptimo.

1.4. Algoritmo propuesto

A continuación, mostramos el código para el algoritmo greedy propuesto:

1.4.1. Código

```python
def greedy(monedas):
    n= len(monedas)
    inicio= 0
    final=n-1

    acciones = [[], []]

    turno = 0 # 0: Sophia, 1: Mateo

    while n > 0:

        if turno == 0:
            if monedas[inicio] > monedas[final]:
                acciones[0].append(monedas[inicio])
                inicio+=1
            else:
                acciones[0].append(monedas[final])
                final-=1

        else:
            if monedas[inicio] > monedas[final]:
                acciones[1].append(monedas[final])
                final-=1
            else:
                acciones[1].append(monedas[inicio])
                inicio+=1
        n-=1

        turno = 1 - turno

    print(f"Ganancia Sofia: {sum(acciones[0])} - Monedas: {acciones[0]} \n")
    print(f"Ganancia Mateo: {sum(acciones[1])} - Monedas: {acciones[1]}")
```

1.4.2. Análisis de la complejidad

- El bucle principal es O(n), ya que procesa cada moneda una vez.
- El cálculo de las sumas al final es O(n).
- No hay operaciones costosas adicionales fuera del bucle.
- Se almacenan listas de acciones.

Por lo tanto, la complejidad temporal total es O(n) y la complejidad espacial es O(n).

1.4.3. Impacto de la variabilidad de valores a la complejidad temporal

La variabilidad de los valores de las monedas no afecta la complejidad temporal, ya que el algoritmo sigue ejecutándose en O(n), independientemente de los valores de las monedas. Esto se debe a que el algoritmo simplemente analiza si un valor es mayor que el otro, y esta operación no se ve influenciada por el número que esté comparando.

1.4.4. Impacto de la variabilidad de valores a la optimalidad

Los valores de las monedas afectan directamente a las decisiones de Sophia, ya que siempre elige la moneda de mayor valor disponible.

Sin embargo, la variabilidad de los valores no afecta la optimalidad del algoritmo, ya que el algoritmo sigue el principio de maximizar el valor de Sophia en cada turno, lo que garantiza el mejor resultado posible para ella.

1.5. Mediciones de tiempo

Para corroborar la complejidad teórica indicada (O(n)), se medirán los tiempos de ejecución del algoritmo greedy planteado con sets de datos generados aleatoriamente con distintos tamaños.

1.5.1. Generador de sets aleatorios

Para generar los sets aleatorios, se utilizó la siguiente función:

```
# Genera n sets de datos de tama o incremental para el problema de las monedas.
# Si se especifica una semilla, se utilizara para generar los datos.
def generar_set_datos_monedas(semilla, n):
    if semilla != None:
        seed(semilla)

    set_datos = []

    for i in range(1, n+1):
        set_datos.append([randint(1, 10000) for _ in range(i * 10)])

    return set_datos
```

De esta forma, se generan n sets de datos que van incrementando su tamaño iterativamente.

1.5.2. Mediciones

Generamos 100 sets de datos, donde cada set incrementa en 10 su tamaño frente al anterior. Las mediciones temporales al ejecutar el algoritmo greedy se muestran a continuación:

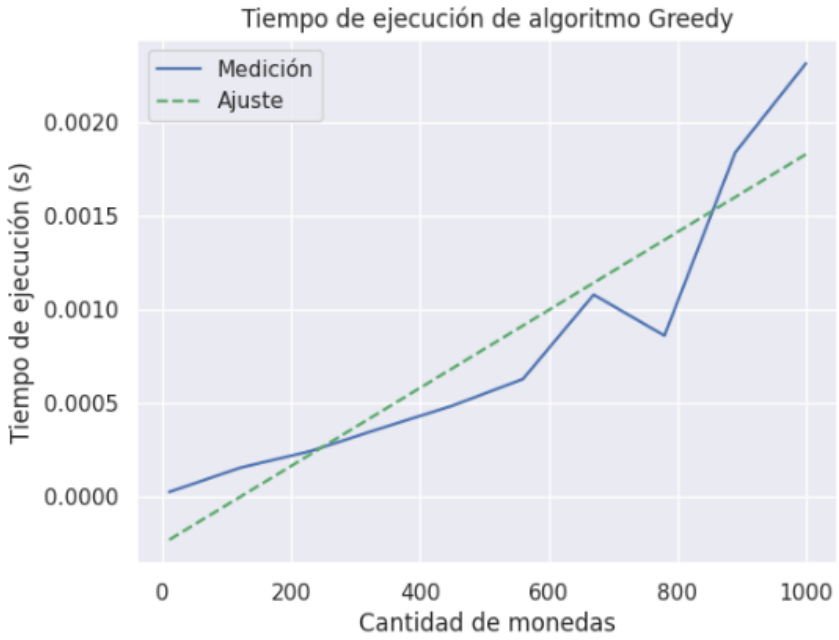

Figura 1: Mediciones temporales para el algoritmo Greedy

Para corroborar la complejidad teórica, se utilizó la técnica de cuadrados mínimos para ajustar los datos medidos a una recta lineal. Notamos que la recta lineal ajusta de manera correcta las mediciones para el algoritmo planteado, por lo que parecería confirmar la complejidad teórica indicada.

Para visualizar mejor el resultado del ajuste, podemos graficar el error absoluto del ajuste segun el tamaño de entrada.

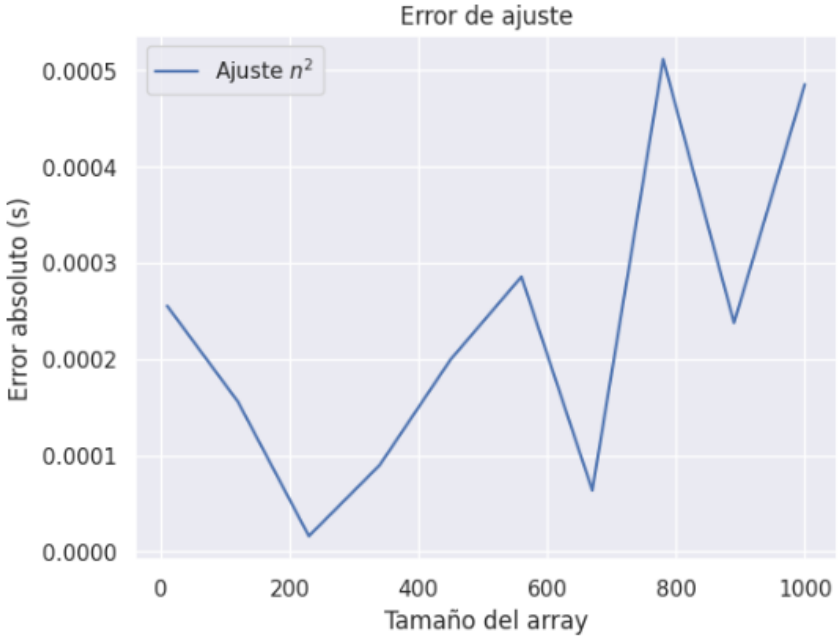

Figura 2: Error absoluto del ajuste según el tamaño de entrada

Notamos que los valores del error absoluto son considerablemente bajos. Esto nos da mas seguridad para afirmar que la complejidad teórica indicada para el algoritmo greedy es correcta.

1.6. Conclusiones

En este primer trabajo práctico se analizó la tecninca Greedy para resolver el problema de las monedas.

Podemos afirmar que:

- El algoritmo greedy propuesto es eficiente y siempre proporciona la solución óptima para Sophia, dado que maximiza su valor en cada turno.
- La complejidad temporal es O(n), lo que lo hace adecuado incluso para listas grandes de monedas.
- La variabilidad de los valores no afecta la eficiencia, pero sí las decisiones de Sophia, lo que puede cambiar la distribución de las monedas elegidas por cada jugador.
- Las mediciones empíricas confirmarán la eficiencia del algoritmo y su comportamiento en diferentes tamaños de entrada.

Este análisis asegura que el algoritmo es eficiente, efectivo y óptimo para el problema planteado.

1.7. Anexo correcciones

1.7.1. Demostración de optimalidad del algoritmo

*Demostración de optimalidad del algoritmo greedy usando inducción fuerte

Definiciones y notación

- Sea $M = [m_1, m_2, \ldots, m_n]$ una lista de monedas, donde m_i es el valor de la i-ésima moneda.
- Sophia usa una estrategia greedy: en su turno, siempre elige la moneda de mayor valor disponible (ya sea m_1 o m_n).
- Mateo usa una estrategia greedy contraria: en su turno, siempre elige la moneda de menor valor disponible.
- $S(n)$: Ganancia total de Sophia para una lista de tamaño n.
- $T(n)$: Ganancia total de Mateo para una lista de tamaño n.

Hipótesis inductiva

Supongamos que para todas las listas de tamaño $k < n$, el algoritmo greedy es óptimo, es decir:

$$\text{Sophia maximiza su ganancia } S(k) \text{ y Mateo minimiza su ganancia } T(k).$$

Caso base

- Para $n = 1$:
$$S(1) = m_1, \quad T(1) = 0.$$
- Para $n = 2$:
$$S(2) = \text{máx}(m_1, m_2), \quad T(2) = \text{mín}(m_1, m_2).$$

Paso inductivo

Consideremos una lista de tamaño n. Queremos demostrar que el algoritmo greedy es óptimo para n, asumiendo que es óptimo para todas las listas de tamaño $k < n$.

*Elección de Sophia En su turno, Sophia elige la moneda de mayor valor entre m_1 y m_n. Supongamos sin pérdida de generalidad que $m_1 \geq m_n$. Entonces, Sophia toma m_1.

*Reducción del problema Después de que Sophia toma m_1, la lista restante es $M' = [m_2, m_3, \ldots, m_n]$, de tamaño $n - 1$.

*Turno de Mateo Mateo ahora elige la moneda de menor valor disponible en M'. Supongamos que elige m_j, donde $m_j = \text{mín}(m_2, m_n)$.

Después de que Mateo toma m_j, la lista restante es M'', de tamaño $n - 2$.

*Aplicación de la hipótesis inductiva Por la hipótesis inductiva, el algoritmo greedy es óptimo para M''. Por lo tanto:

$$S(n-2) = \text{Ganancia de Sophia en } M'', \quad T(n-2) = \text{Ganancia de Mateo en } M''.$$

*Ganancia total de Sophia La ganancia total de Sophia es:

$$S(n) = m_1 + S(n-2).$$

*Ganancia total de Mateo La ganancia total de Mateo es:

$$T(n) = m_j + T(n-2).$$

Conclusión

Por el principio de inducción fuerte, el algoritmo greedy es óptimo para cualquier tamaño de lista n. La elección greedy de Sophia en cada paso maximiza su ganancia total, y la hipótesis inductiva garantiza que esto se cumple para todas las listas más pequeñas.

1.7.2. Impacto de la variabilidad de valores a la complejidad algorítmica—

Para analizar el impacto de la variabilidad de los valores de monedas en el tiempo de ejecución de nuestro algoritmo greedy, se medirán los tiempos de ejecución del algoritmo como se hizo en el apartado anterior, pero esta vez comparando para sets de datos de baja variabilidad (valores de 1 a 100) y de alta variabilidad (valores de 1 a 10000).

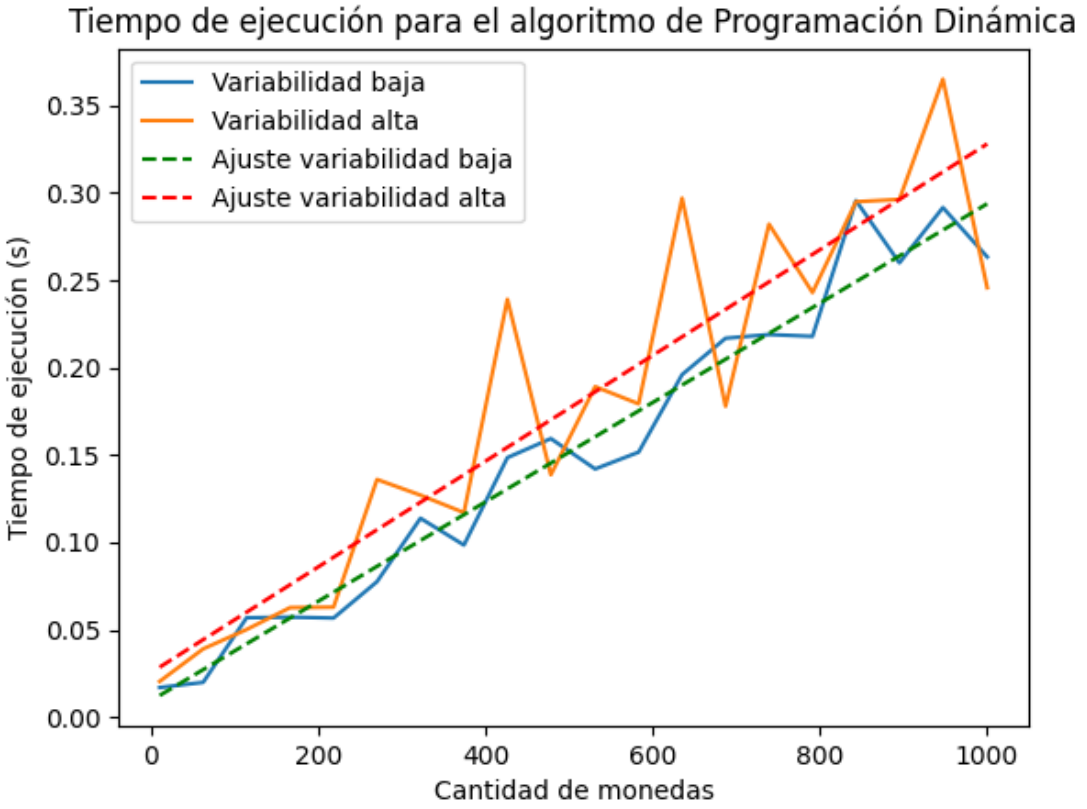

Figura 3: Gráfico comparativo entre los tiempos de ejecución con variabilidad baja y alta

Vemos como los tiempos de ejecución para el algoritmo greedy con valores con variabilidad alta es ligeramente superior a los de variabilidad baja. Sin embargo, consideramos que no es significativo el impacto de los valores a la complejidad algoritmica, ya que el aumento del tiempo fue muy bajo con respecto al aumento de la variabilidad.

2. PARTE 2

2.1. Introducción

2.1.1. Contexto

Pasan los años. Mateo ahora tiene 7 años. Los mismos años que tenía Sophia cuando comenzaron a jugar al juego de las monedas. Eso quiere decir que Mateo también ya aprendió sobre algoritmos greedy, y lo comenzó a aplicar. Esto hace que ahora quién gane dependa más de quién comience y un tanto de suerte.

Esto no le gusta nada a Sophia. Ella quiere estar segura de ganar siempre. Lo bueno es que ella comenzó a aprender sobre programación dinámica. Ahora va a aplicar esta nueva técnica para asegurarse ganar siempre que pueda

2.1.2. Consigna

1. Hacer un análisis del problema, plantear la ecuación de recurrencia correspondiente y proponer un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado: Dada la secuencia de monedas $m_1, m_2, ..., m_n$, sabiendo que Sophia empieza el juego y que Mateo siempre elegirá la moneda más grande para sí entre la primera y la última moneda en sus respectivos turnos, definir qué monedas debe elegir Sophia para asegurarse obtener el máximo valor acumulado posible. Esto no necesariamente le asegurará a Sophia ganar, ya que puede ser que esto no sea obtenible, dado por cómo juega Mateo. Por ejemplo, para [1, 10, 5], no importa lo que haga Sophia, Mateo ganará.

2. Demostrar que la ecuación de recurrencia planteada en el punto anterior en efecto nos lleva a obtener el máximo valor acumulado posible.

3. Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores de las monedas.

4. Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también.

5. Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración (generando sus propios sets de datos). Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos. Para esto, proveemos una explicación detallada, en conjunto de ejemplos

6. Agregar cualquier conclusión que parezca relevante.

2.2. Análisis del problema

2.2.1. Descripción de problema

Sofía y Mateo están jugando a un juego de monedas. La secuencia de monedas es $m_1, m_2, \ldots, m_n$. En cada turno, Sofía o Mateo eligen una moneda de una fila de monedas que está a la izquierda o a la derecha. El objetivo de Sofía es maximizar su valor acumulado. Sin embargo, Mateo, que siempre juega con la estrategia más agresiva, siempre tomará la moneda más grande de las dos opciones disponibles (de la izquierda o de la derecha) en su turno.

Sofía comienza primero. Aunque Sofía intente maximizar su valor, la estrategia de Mateo puede hacer que, en algunos casos, ella no pueda ganar. Por ejemplo, en una secuencia de monedas como $[1, 10, 5]$, no importa lo que haga Sofía, Mateo ganará debido a la forma en que elige las monedas. Este planteo busca que por cada moneda que tenga la opcion de elegit, poder analizar como impactara eso en toda la jugada global, es decir, resolviendo los subproblemas que se desprenden de cada posible decisión.

2.2.2. Propuesta de algoritmo dinamico

Nuestro objetivo es poder adelantarnos a la posible futura jugada de Mateo, ya que, siempre elegira la mejor opcion localmente, pero Sofia puede adelantarse usando programación dinamica y ver que monedas son mas beneficiosas y asegurarse de no dejarselas disponibles para Mateo.

Ejemplo: Si tuvieran las siguientes monedas n= [70, 80, 27, 30]

Sofia en este caso tiene que poder adelantarse estrategicamente hablando y hacer el correcto analisis de las que estan disponible en su turno y las que estaran disponibles en los siguientes.

En este caso, como Sofia es mas grande y mas inteligente usara el razonamiento para darse cuenta que si elige la moneda[0]= 70 en lugar de moneda[-1]=30 eso le traera un problema a futuro cuando Mateo en su turno aplique su estrategia y se lleve la moneda mas grande de toda la partida (80).

Con eleccion codiciosa el resultado seria:

Monedas Sofia= 70, 30. Suma= 100

Monedas Mateo= 80, 27. Suma= 107

Con eleccion dinamica el resultado se veria asi:

Monedas Sofia= 30, 80. Suma= 110

Monedas Mateo= 70, 27. Suma= 97

Es decir, Sofia se adelanto a la siguiente jugada.

La estrategia es no tomar una decision como en la partida anterior (version codiciosa) solo teniendo en cuenta las 2 monedas actuales disponibles sino que pensar en las posibles monedas que estaran disponibles en los siguientes turnos.

Ecuación de Recurrencia

Definimos $dp[i][j]$ como el valor máximo que Sofía puede obtener si juega con las monedas en el rango de índices $[i, j]$, es decir, si tiene que tomar una decisión sobre qué monedas seleccionar entre m_i y m_j.

Caso base Cuando solo queda una moneda en el rango, el valor máximo es simplemente el valor de esa moneda:

$$dp[i][i] = m_i$$

Caso recursivo Cuando hay más de una moneda en el rango, Sofía tiene dos opciones: elegir la primera moneda m_i o la última m_j. Sin embargo, debe tener en cuenta que Mateo, en su turno, siempre tomará la moneda más grande entre las dos opciones disponibles (de la izquierda o de la

derecha).

- Si Sofía elige m_i, el siguiente turno es de Mateo. Mateo entonces elegirá entre m_{i+1} y m_j, y el valor total que Sofía podrá obtener será el valor de m_i más el valor restante de las monedas después de que Mateo elija. - Si Sofía elige m_j, el siguiente turno es de Mateo, quien elegirá entre m_i y m_{j-1}, y el valor total que Sofía podrá obtener será el valor de m_j más el valor restante después de que Mateo elija.

Entonces, la ecuación de recurrencia es:

$$dp[i][j] = \text{máx}\left(m_i + \text{máx}(dp[i+2][j], dp[i+1][j-1]), m_j + \text{máx}(dp[i+1][j-1], dp[i][j-2])\right)$$

La idea detrás de la recurrencia es que Sofía busca maximizar su valor actual, pero debe anticipar las elecciones de Mateo, por lo que la estrategia es elegir la opción que deje a Mateo con las peores opciones posibles (es decir, la que minimice lo que Mateo puede obtener después de su turno).

2.3. Demostración de ecuación de recurrencia

Para demostrar que la ecuación de recurrencia planteada nos lleva a obtener el máximo valor acumulado posible, debemos realizar un análisis basado en la estructura de las decisiones de Sofía y Mateo, y cómo estas afectan a la función $dp[i][j]$.

2.3.1. Planteamiento de la Ecuación de Recurrencia

La ecuación de recurrencia planteada es la siguiente:

$$dp[i][j] = \text{máx}\left(m_i + \text{máx}(dp[i+2][j], dp[i+1][j-1]), m_j + \text{máx}(dp[i+1][j-1], dp[i][j-2])\right)$$

Donde:

- $dp[i][j]$ es el valor máximo que Sofía puede obtener al jugar con las monedas en el rango de índices $[i, j]$.
- m_i y m_j son las monedas en las posiciones i y j, respectivamente.
- La función $\text{máx}(dp[i+2][j], dp[i+1][j-1])$ representa tambien el valor que puede obtener Sofía si elige m_i, ya que después de su elección, Mateo puede elegir entre las opciones $[i+2, j]$ o $[i+1, j-1]$ para minimizar la ganancia de Sofía.
- La función $\text{máx}(dp[i+1][j-1], dp[i][j-2])$ representa el valor que puede obtener Sofía si elige m_j, ya que Mateo puede elegir entre las opciones $[i+1, j-1]$ o $[i, j-2]$ para minimizar la ganancia de Sofía.

2.3.2. Demostración por Inducción

Para demostrar que esta ecuación lleva efectivamente al valor óptimo, utilizaremos inducción sobre el tamaño del subproblema, es decir, sobre la longitud del rango de monedas considerado.

Caso base

Cuando hay solo una moneda en el rango $[i, i]$, es claro que Sofía tomará esa moneda, ya que no hay otra opción. En este caso, tenemos:

$$dp[i][i] = m_i$$

Este es el caso base, que está de acuerdo con la ecuación de recurrencia planteada, ya que no hay decisiones posteriores para tomar.

Paso inductivo

Supongamos que la ecuación de recurrencia es válida para rangos de monedas más pequeños. Ahora, consideremos un rango de longitud mayor, $[i, j]$, con $j > i$.

Sofía tiene dos opciones:

1. **Elegir la moneda m_i:** En este caso, Mateo elegirá entre dos rangos: $[i+2, j]$ o $[i+1, j-1]$. La elección de Mateo afectará el valor total de las monedas restantes. La ecuación de recurrencia toma el valor máximo entre estas dos opciones para asegurar que Sofía obtenga el mejor resultado posible en este escenario. Luego, Sofía suma el valor de m_i a la ganancia que Mateo podría dejarle, lo cual se expresa como:

$$m_i + \text{máx}(dp[i+2][j], dp[i+1][j-1])$$

2. **Elegir la moneda m_j:** En este caso, Mateo elegirá entre los rangos $[i+1, j-1]$ o $[i, j-2]$. De manera similar, la ecuación de recurrencia calcula la ganancia para Sofía después de la elección de Mateo, y Sofía suma el valor de m_j. Esto se expresa como:

$$m_j + \text{máx}(dp[i+1][j-1], dp[i][j-2])$$

Finalmente, Sofía seleccionará la opción que maximice su ganancia:

$$dp[i][j] = \text{máx}\left(m_i + \text{máx}(dp[i+2][j], dp[i+1][j-1]), m_j + \text{máx}(dp[i+1][j-1], dp[i][j-2])\right)$$

2.3.3. Correctitud de la Ecuación

La ecuación de recurrencia tiene en cuenta las decisiones de Sofía y Mateo de manera adecuada. Sofía busca maximizar su valor total, pero debe anticipar las elecciones de Mateo, quien siempre elegirá la opción que minimice la ganancia de Sofía. Al elegir la moneda más grande disponible, Mateo sigue una estrategia agresiva que minimiza la opción de Sofía. La ecuación asegura que Sofía tome la mejor decisión posible en cada caso, maximizando su ganancia acumulada.

2.3.4. Conclusión

La ecuación de recurrencia planteada es correcta porque sigue la lógica del juego y permite calcular el valor máximo que Sofía puede obtener. Al considerar las decisiones de ambos jugadores, y al calcular el valor máximo para cada subproblema de manera recursiva, garantizamos que se obtiene el valor óptimo para Sofía al final del juego. La solución se construye correctamente utilizando la programación dinámica y la ecuación de recurrencia permite obtener la máxima ganancia acumulada posible para Sofía en cualquier secuencia de monedas.

2.4. Algoritmo en programación dinámica

El algoritmo sigue la idea de llenar una tabla dp donde cada celda $dp[i][j]$ almacena el valor máximo que Sofía puede obtener al jugar con las monedas entre m_i y m_j. La solución óptima será el valor almacenado en $dp[0][n-1]$, donde n es el número total de monedas.

2.4.1. Código

```
def pd(monedas):
    n = len(monedas)
    # Crear tabla bidimensional para almacenar los resultados
    dp = [[0] * n for _ in range(n)]

    # Llenar la tabla base: casos con una sola moneda
    for i in range(n):
        dp[i][i] = monedas[i]

    # Llenar la tabla para rangos crecientes de tama o
    for longitud in range(2, n + 1):    # Tama o del subarreglo
        for inicio in range(n - longitud + 1):  # Inicio del rango
            fin = inicio + longitud - 1         # Fin del rango

            # Calcular la mejor elecci n para el rango [inicio, fin]
            elegir_inicio = monedas[inicio] + max(
                dp[inicio + 2][fin] if (inicio + 2 <= fin) and (monedas[inicio+1]>
    monedas[fin]) else 0,
                dp[inicio + 1][fin - 1] if (inicio ++ 1 <= fin - 1) and (monedas[
    inicio+1]<monedas[fin])else 0,
            )
            elegir_fin = monedas[fin] + max(
                dp[inicio][fin - 2] if (inicio <= fin - 2) and (monedas[inicio]<
    monedas[fin-1])  else 0,
                dp[inicio + 1][fin - 1] if (inicio + 1) <= fin - 1 and (monedas[
    inicio]>monedas[fin-1]) else 0,
            )
            dp[inicio][fin] = max(elegir_inicio,  elegir_fin )

    # El resultado est  en dp[0][n-1], considerando todas las monedas

    print(f"Ganancia Sofia: {dp[0][n-1]} \nGanancia Mateo: {sum(monedas)-dp[0][n
    -1]}")
```

2.4.2. Análisis de la complejidad

- Llenamos la tabla para rangos crecientes de tamaño en $O(n^2)$.
- Calculamos la mejor elección para el rango [inicio, fin] usando la tabla, accediendo a ella en $O(1)$.
- Comparamos las decisiones en $O(1)$.
- Se guarda la mejor opción y la decisión tomada en la tabla en $0(1)$.
- La tabla dp es una matriz bidimensional de tamaño nxn lo que requiere $O(n^2)$ de espacio.

Por lo tanto, la complejidad temporal total es $O(n^2)$ y la complejidad espacial es $O(n^2)$.

2.4.3. Análisis del impacto de la variabilidad de los valores de las monedas en los tiempos del algoritmo

El tiempo de ejecución del algoritmo **no depende directamente de los valores de las monedas**, sino de la cantidad de monedas (n) debido a las siguientes razones:

Acceso constante a las tablas `dp` y `decisiones` El algoritmo llena la tabla `dp` realizando operaciones en tiempo constante $O(1)$ para cada combinación de $inicio$ y fin. Las comparaciones y selecciones de los valores máximos dependen de los índices y no del tamaño o magnitud de los valores numéricos de las monedas.

Impacto de las decisiones de Mateo Aunque Mateo selecciona la moneda más grande entre las disponibles (primera o última), esta decisión implica una comparación en $O(1)$. La variabilidad en los valores de las monedas afecta **qué camino toma el algoritmo**, pero no aumenta la cantidad de operaciones realizadas. Independientemente de los valores, el número de iteraciones en los bucles es fijo y depende solo de n.

Relevancia de la distribución de valores Si los valores de las monedas están distribuidos de forma uniforme o si hay grandes disparidades, las elecciones de Mateo y Sophia podrían ser más evidentes o requerir más análisis manual, pero **esto no afecta la complejidad computacional**. El algoritmo siempre realiza las mismas comparaciones en cada rango definido por $inicio$ y fin, independientemente de la magnitud o distribución de los valores.

Conclusión La variabilidad de los valores de las monedas **no afecta los tiempos del algoritmo en términos de complejidad asintótica**, ya que todas las operaciones involucradas (comparaciones y actualizaciones de tablas) son independientes de los valores específicos.

Sin embargo, en aplicaciones prácticas, una mayor variabilidad en los valores podría influir en el resultado de las elecciones y en la percepción de eficiencia, pero no en la cantidad de operaciones realizadas.

Por lo tanto, la complejidad temporal y espacial del algoritmo sigue siendo $O(n^2)$, sin importar cómo varíen los valores de las monedas.

2.5. Análisis temporal

Para corroborar la complejidad teórica indicada ($O(n^2)$), se medirán los tiempos de ejecución del algoritmo de programación dinámica planteado con sets de datos generados aleatoriamente con distintos tamaños.

2.5.1. Generador de sets aleatorios

Para generar los sets aleatorios, se utilizó la siguiente función:

```
# Genera n sets de datos de tamano incremental para el problema de las monedas.
# Si se especifica una semilla, se utilizara para generar los datos.
def generar_set_datos_monedas(semilla, n):
    if semilla != None:
        seed(semilla)

    set_datos = []

    for i in range(1, n+1):
        set_datos.append([randint(1, 10000) for _ in range(i * 10)])

    return set_datos
```

De esta forma, se generan n sets de datos que van incrementando su tamaño iterativamente.

2.5.2. Mediciones

Generamos 100 sets de datos, donde cada set incrementa en 10 su tamaño frente al anterior. Las mediciones temporales al ejecutar el algoritmo de programación dinámica se muestran a continuación:

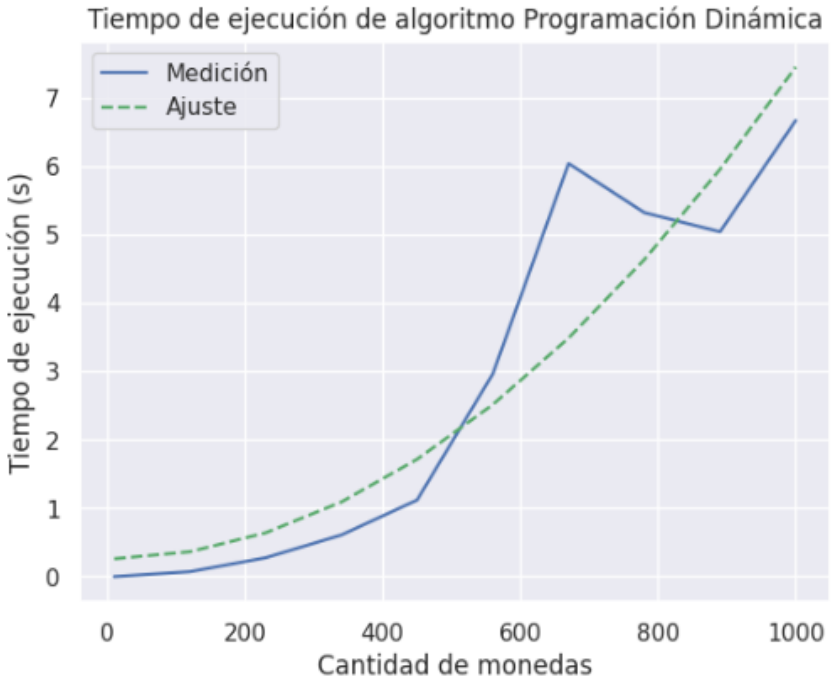

Figura 4: Mediciones temporales para el algoritmo planteado

Para corroborar la complejidad teórica, se utilizó la técnica de cuadrados mínimos para ajustar los datos medidos a una recta cuadrática. Podemos ver como la recta de ajuste cuadrático ajusta

en terminos generales de manera adecuada a las mediciones, por lo que parecería confirmar que la complejidad teórica calculada es correcta.

Para visualizar mejor el resultado del ajuste, podemos graficar el error absoluto del ajuste segun el tamaño de entrada.

Figura 5: Error absoluto del ajuste según el tamaño de entrada

En este caso, notamos que el error absoluto para el ajuste si fue de un valor considerable. Podemos ver como en términos generales el error absoluta ronda el valor de 1.0, sin embargo para ciertos tamaños de entradas tiene un pico en 2.5. Esto coincide con el gráfico de mediciones ya que en el mismo intervalo el ajuste se desvía más de la medición. Entendemos que estos son casos particulares y que, en general, el ajuste fue bueno y por lo tanto la complejidad teórica indicada es correcta.

2.6. Conclusiones

El algoritmo planteado utiliza programación dinámica para asegurar que Sofía maximice su valor acumulado, anticipando las decisiones de Mateo. La complejidad temporal del algoritmo es $O(n^2)$, y la espacial es también $O(n^2)$, debido a las tablas *dp* y *decisiones*. La estrategia optimiza las elecciones de Sofía, aunque en algunos casos puede que ella no gane debido a la forma en que Mateo elige sus monedas.

2.7. Anexo Correcciones

2.7.1. Ecuacion de recurrencia

El problema consiste en maximizar la ganancia de Sofía en un juego de elección de monedas entre dos jugadores (Sofía y Mateo). Para resolverlo, se utiliza programación dinámica, donde se define una tabla $dp[i][j]$ que representa la máxima ganancia que puede obtener Sofía en el rango de monedas $[i, j]$.

La ecuación de recurrencia se define de la siguiente manera:

$$dp[i][j] = \begin{cases} \text{máx}\left(\begin{cases} monedas[i] + \text{mín}(dp[i+2][j], dp[i+1][j-1]) \\ monedas[j] + \text{mín}(dp[i+1][j-1], dp[i][j-2]) \end{cases}\right) & \text{si } i < j \\ monedas[i] & \text{si } i = j \\ 0 & \text{si } i > j \end{cases}$$

Explicación de la ecuación

- **Caso base** $(i = j)$: Si solo hay una moneda en el rango $[i, j]$, Sofía toma esa moneda, y la ganancia es $monedas[i]$.
- **Caso base** $(i > j)$: Si no hay monedas en el rango $[i, j]$, la ganancia es 0.
- **Caso general** $(i < j)$: Sofía tiene dos opciones:
  - Tomar la moneda en la posición i: Luego, Mateo elegirá la mejor opción para él, y Sofía se quedará con el mínimo entre $dp[i+2][j]$ y $dp[i+1][j-1]$.
  - Tomar la moneda en la posición j: Luego, Mateo elegirá la mejor opción para él, y Sofía se quedará con el mínimo entre $dp[i+1][j-1]$ y $dp[i][j-2]$.

  Sofía elige la opción que maximiza su ganancia.

2.7.2. Demostración de la Ecuacion de Recurrencia

Para demostrar que la ecuación de recurrencia lleva a una solución óptima, utilizaremos inducción matemática.

Caso base

- Si $i = j$, solo hay una moneda, y Sofía la toma. Esto es óptimo porque no hay otras opciones.
- Si $i > j$, no hay monedas, y la ganancia es 0. Esto también es óptimo.

Hipótesis inductiva

Supongamos que para todos los rangos $[i', j']$ más pequeños que $[i, j]$, la ecuación de recurrencia calcula correctamente la ganancia óptima de Sofía.

Paso inductivo

Para el rango $[i, j]$, Sofía tiene dos opciones:

- Tomar la moneda en la posición i: Luego, Mateo elegirá la mejor opción para él, y Sofía se quedará con el mínimo entre $dp[i+2][j]$ y $dp[i+1][j-1]$. Por la hipótesis inductiva, estos valores ya son óptimos.
- Tomar la moneda en la posición j: Luego, Mateo elegirá la mejor opción para él, y Sofía se quedará con el mínimo entre $dp[i+1][j-1]$ y $dp[i][j-2]$. Por la hipótesis inductiva, estos valores ya son óptimos.

Sofía elige la opción que maximiza su ganancia, lo que garantiza que la solución es óptima para el rango $[i, j]$.

Conclusión

Por inducción, la ecuación de recurrencia calcula correctamente la ganancia óptima de Sofía para cualquier rango $[i, j]$, lo que demuestra que lleva a una solución óptima.

2.7.3. Algoritmo en Programacion Dinámica

Código El algoritmo sigue la idea de llenar una tabla dp donde cada celda $dp[i][j]$ almacena el valor máximo que Sofía puede obtener al jugar con las monedas entre m_i y m_j. La solución óptima será el valor almacenado en $dp[0][n-1]$, donde n es el número total de monedas.

```
def pd(monedas):
    n = len(monedas)
    # Crear tabla bidimensional para almacenar los resultados
    dp = [[0] * n for _ in range(n)]

    # Llenar la tabla base: casos con una sola moneda
    for i in range(n):
        dp[i][i] = monedas[i]

    # Llenar la tabla para rangos crecientes de tama o
    for longitud in range(2, n + 1):            # Tama o del subarreglo
        for inicio in range(n - longitud + 1):  # Inicio del rango
            fin = inicio + longitud - 1         # Fin del rango

            # Calcular la mejor elecci n para el rango [inicio, fin]

            # Opcion inicio
            if inicio + 1 <= fin:
                # Considero las opciones de Mateo
                if monedas[inicio + 1] >= monedas[fin]:
                    elegir_inicio = monedas[inicio] + dp[inicio + 2][fin] if inicio + 2 <= fin else monedas[inicio]
                else:
                    elegir_inicio = monedas[inicio] + dp[inicio + 1][fin - 1] if inicio + 1 <= fin - 1 else monedas[inicio]
            else:
                elegir_inicio = monedas[inicio]

            # Opcion Final
            if inicio <= fin - 1:
                # Considero las opciones de Mateo
                if monedas[inicio] >= monedas[fin - 1]:
                    elegir_fin = monedas[fin] + dp[inicio + 1][fin - 1] if inicio + 1 <= fin - 1 else monedas[fin]
                else:
                    elegir_fin = monedas[fin] + dp[inicio][fin - 2] if inicio <= fin - 2 else monedas[fin]
            else:
                elegir_fin = monedas[fin]

            dp[inicio][fin] = max(elegir_inicio, elegir_fin)

    # El resultado est  en dp[0][n-1], considerando todas las monedas

    print(f"Ganancia Sofia: {dp[0][n-1]} \nGanancia Mateo: {sum(monedas)-dp[0][n-1]}")
```

Análisis de complejidad El algoritmo utiliza programación dinámica para resolver el problema de maximizar la ganancia de Sofía en un juego de elección de monedas. La complejidad temporal se desglosa de la siguiente manera:

Complejidad Temporal

- **Llenar la tabla para rangos crecientes de tamaño**:
  - Se utilizan dos bucles anidados para llenar la tabla dp.
  - El bucle externo itera sobre los tamaños de los subarreglos (desde 2 hasta n), lo que toma $O(n)$.
  - El bucle interno itera sobre los índices de inicio de los subarreglos, lo que también toma $O(n)$.
  - Por lo tanto, la complejidad de estos bucles anidados es $O(n^2)$.

- **Calcular la mejor elección para el rango [inicio, fin]**:
  - Se accede a la tabla dp en tiempo constante $O(1)$ para obtener los valores de $dp[i+2][j]$, $dp[i+1][j-1]$, etc.
  - Las comparaciones y operaciones matemáticas también se realizan en $O(1)$.
- **Guardar la mejor opción en la tabla**:
  - Una vez calculada la mejor opción, se guarda en la tabla $dp[inicio][fin]$ en $O(1)$.
- **Complejidad temporal total**:
  - Dado que el paso dominante es el llenado de la tabla con los bucles anidados, la complejidad temporal total es $O(n^2)$.

Complejidad Espacial

La complejidad espacial del algoritmo se debe principalmente a la tabla dp:

- **Tabla** dp:
  - La tabla dp es una matriz bidimensional de tamaño $n \times n$, donde n es el número de monedas.
  - Esto requiere $O(n^2)$ de espacio.
- **Variables adicionales**:
  - Las variables temporales como $elegir\_inicio$ y $elegir\_fin$ ocupan espacio constante $O(1)$.
  - No hay estructuras de datos adicionales que crezcan con el tamaño de la entrada.
- **Complejidad espacial total**:
  - La complejidad espacial está dominada por la tabla dp, por lo que es $O(n^2)$.

Análisis del Impacto de la Variabilidad de los Valores de las Monedas

El tiempo de ejecución del algoritmo **no depende directamente de los valores de las monedas**, sino de la cantidad de monedas (n) debido a las siguientes razones:

- Acceso Constante a las Tablas dp y Decisiones

El algoritmo llena la tabla dp realizando operaciones en tiempo constante $O(1)$ para cada combinación de $inicio$ y fin. Las comparaciones y selecciones de los valores máximos dependen de los índices y no del tamaño o magnitud de los valores numéricos de las monedas.

- Impacto de las Decisiones de Mateo

Aunque Mateo selecciona la moneda más grande entre las disponibles (primera o última), esta decisión implica una comparación en $O(1)$. La variabilidad en los valores de las monedas afecta qué **camino toma el algoritmo**, pero no aumenta la cantidad de operaciones realizadas. Independientemente de los valores, el número de iteraciones en los bucles es fijo y depende solo de n.

- Relevancia de la Distribución de Valores

Si los valores de las monedas están distribuidos de forma uniforme o si hay grandes disparidades, las elecciones de Mateo y Sofía podrían ser más evidentes o requerir más análisis manual, pero esto **no afecta la complejidad computacional**. El algoritmo siempre realiza las mismas comparaciones en cada rango definido por $inicio$ y fin, independientemente de la magnitud o distribución de los valores.

- Conclusión

La variabilidad de los valores de las monedas **no afecta los tiempos del algoritmo en términos de complejidad asintótica**, ya que todas las operaciones involucradas (comparaciones y actualizaciones de tablas) son independientes de los valores específicos.

Sin embargo, en aplicaciones prácticas, una mayor variabilidad en los valores podría influir en el resultado de las elecciones y en la percepción de eficiencia, pero no en la cantidad de operaciones realizadas.

Por lo tanto, la complejidad temporal y espacial del algoritmo sigue siendo $O(n^2)$, sin importar cómo varíen los valores de las monedas.

2.7.4. Mediciones

Se nos pidió mejorar la forma de medir el error. Para eso, realizamos 5 ejecuciónes de distintos sets de datos con el mismo tamaño, para luego tomar el promedio de las mediciones. De esta forma, obtenemos un valor más cercano a la realidad que nos permetirá medir de mejor manera el error.

Recordamos que cada set de datos se conforma de 100 arreglos de monedas, cada uno incrementando su tamaño en 10 respecto al anterior.

A continuación, mostramos las mediciones obtenidas:

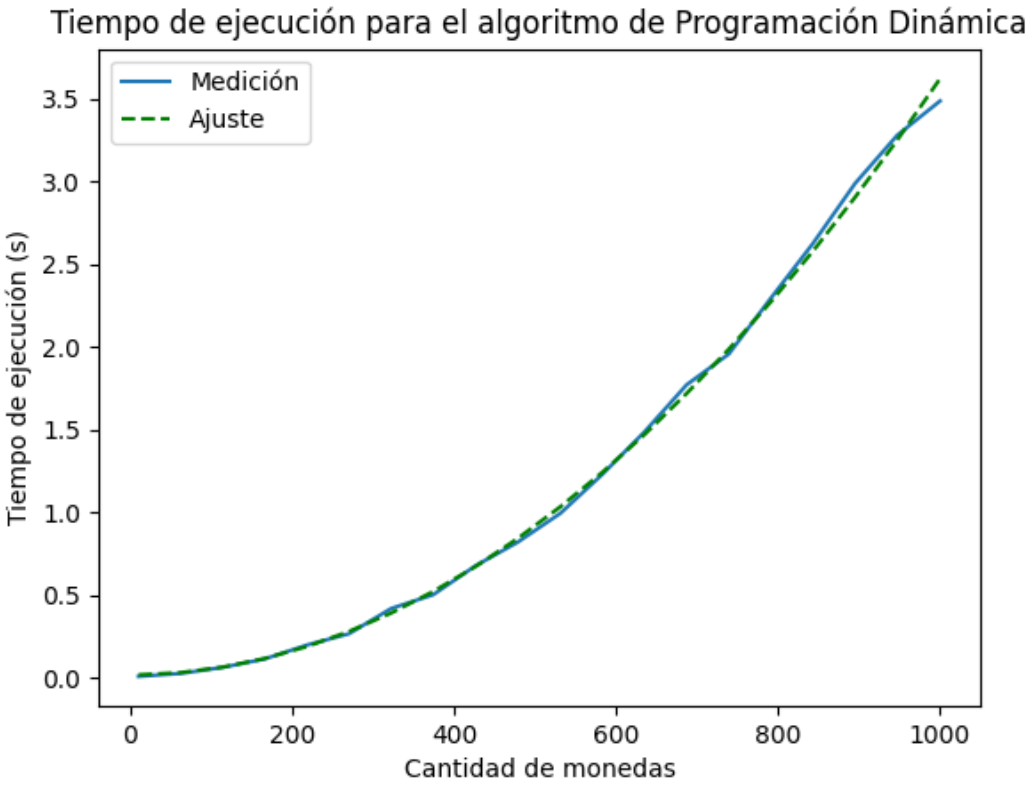

Figura 6: Tiempos de ejecución para el algoritmo de Programación Dinámica

Se utilizó la técnica de cuadrados mínimos para corroborar la complejidad teórica indicada. Se puede ver cómo la recta de ajuste cuadrático ajusta casi de manera perfecta para las mediciones obtenidas, dandonos seguridad para afirmar que la complejidad de $O(n^2)$.

Para visualizar mejor el resultado del ajuste, graficamos el error absoluto del ajuste segun la cantidad de monedas:

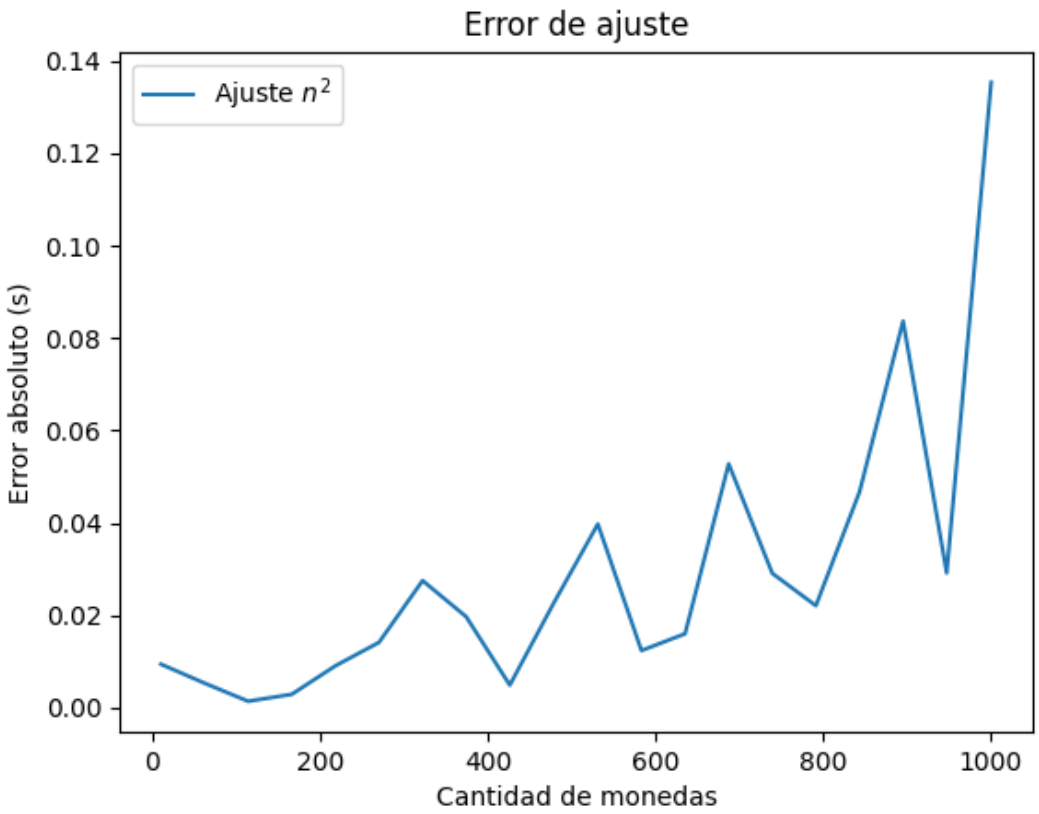

Figura 7: Enter Caption

Como vemos, el error absoluto obtenido es muy pequeño con respecto al que obtuvimos en la primera entrega, por lo que entendemos que efectivamente se pudo mejorar la forma de medir el error.

3. PARTE 3

3.1. Introducción

3.1.1. Contexto

Los hermanos siguieron creciendo. Mateo también aprendió sobre programación dinámica, y cada uno aplicaba la lógica sabiendo que el otro también lo hacía. El juego de las monedas se tornó aburrido en cuánto notaron que siempre ganaba quien empezara, o según la suerte. Los años pasaron, llegó la adolescencia y empezaron a tener gustos diferentes. En general, jugaban a juegos individuales. En particular, Sophia estaba muy enganchada con un juego inventado en Argentina por Jaime Poniachik (uno de los fundadores de Ediciones de Mente) en 1982: La Batalla Naval Individual.

En dicho juego, tenemos un tablero de nxm casilleros, y k barcos. Cada barco i tiene b_i de largo. Es decir, requiere de b_i casilleros para ser ubicado. Todos los barcos tienen 1 casillero de ancho. El tablero a su vez tiene un requisito de consumo tanto en sus filas como en sus columnas. Si en una fila indica un 3, significa que deben haber 3 casilleros de dicha fila siendo ocupados. Ni más, ni menos. No podemos poner dos barcos de forma adyacente (es decir, no pueden estar contiguos ni por fila, ni por columna, ni en diagonal directamente). Debemos ubicar todos los barcos de tal manera que se cumplan todos los requisitos. A continuación mostramos un ejemplo de un juego resuelto:

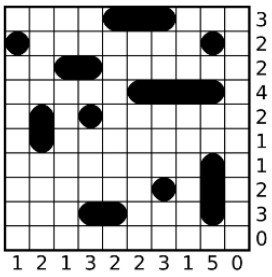

Figura 8: Ejemplo de Batalla Naval Individual resuelta.

3.1.2. Consigna

Para los primeros dos puntos considerar la versión de decisión del problema de La Batalla Naval: Dado un tablero de nxm casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo), una lista de restricciones para las filas (donde la restricción j corresponde a la cantidad de casilleros a ser ocupados en la fila j) y una lista de restricciones para las columnas (símil filas, pero para columnas), ¿es posible definir una ubicación de dichos barcos de tal forma que se cumplan con las demandas de cada fila y columna, y las restricciones de ubicación?

1. Demostrar que el Problema de la Batalla Naval se encuentra en NP.
2. Demostrar que el Problema de la Batalla Naval es, en efecto, un problema NP-Completo. Si se hace una reducción involucrando un problema no visto en clase, agregar una (al menos resumida) demostración que dicho problema es NP-Completo.
3. Escribir un algoritmo que, por backtracking, obtenga la solución óptima al problema (valga la redundancia) en la versión de optimización: Dado un tablero de nxm casilleros, y una lista de k barcos (donde el barco i tiene b_i de largo) una lista de las demandas de las n filas y una lista de las m demandas de las columnas, dar la asignación de posiciones de los barcos de tal forma que se reduzca al mínimo la cantidad de demanda incumplida. Pueden no utilizarse todos los barcos. Si simplemente no se cumple que una columna que debería tene 3 casilleros

ocupados tiene 1, entonces contará como 2 de demanda incumplida. Por el contrario, no está permitido exceder la cantidad demandada. Generar sets de datos para corroborar su correctitud, así como tomar mediciones de tiempos.

4. Escribir un modelo de programación lineal que resuelva el problema de forma óptima. Ejecutarlo para los mismos sets de datos para corroborar su correctitud. Tomar mediciones de tiempos y compararlas con las del algoritmo que implementa Backtracking.

5. John Jellicoe (almirante de la Royal Navy durante la batalla de Jutlandia) nos propone el siguiente algoritmo de aproximación: Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.

   Este algoritmo sirve como una aproximación para resolver el problema de La Batalla Naval. Implementar dicho algoritmo, analizar su complejidad y analizar cuán buena aproximación es. Para esto, considerar lo siguiente: Sea I una instancia cualquiera del problema de La Batalla Naval, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leqslant r(A)$ para todas las instancias posibles. Calcular $r(A)$ para el algoritmo dado, demostrando que la cota está bien calculada. Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar dicha relación. Realizar también mediciones que contemplen volúmenes de datos ya inmanejables para el algoritmo exacto, a fin de corroborar empíricamente la cota calculada anteriormente.

6. Opcional: Implementar alguna otra aproximación (o algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto anterior. Indicar y justificar su complejidad. No es obligatorio hacer este punto para aprobar el trabajo práctico (pero si resta puntos no hacerlo).

7. Agregar cualquier conclusión que parezca relevante.

3.2. Demostración NP

Para demostrar que la Batalla Naval Individual se encuentra en NP, debemos demostrar que existe un certificador eficiente, es decir, que debemos poder validar en tiempo polinomial que una solución dada cumple con las restricciones del problema.

Estas restricciones son:

- No existen barcos ubicados de forma diagonal.
- Se cumple con las demandas de fila.
- Se cumple con las demandas de columna
- Todos los barcos pedidos estan ubicados en el tablero

Además, representamos a los barcos en el tablero como una serie de 1s contiguos (tanto verticales como horizontales), y al agua en el tablero con 0.

3.2.1. Algoritmo certificador eficiente

A continuación se muestra el código del certificador eficiente planteado. El certificador recibe:

- El tablero de $n \times m$ casilleros
- La lista de k barcos, donde el barco i tiene b_i de largo
- La lista de restricciones para las filas, donde la restriccion j corresponde a la cantidad de casilleros a ser ocupados por la fila j
- La lista de restricciones para las columnas, con el mismo formato que para las filas.

```
def certificador_eficiente(tablero, barcos, requisitos_fil, requisitos_col):

    # Chequeo barcos diagonales -> O(nxm)
    if chequeo_barcos_diagonales(tablero) == False:
        return False

    # Chequeo requisitos de fila -> O(nxm)
    for i in range(len(tablero)):
        if tablero[i].count(1) != requisitos_fil[i]:
            return False

    # Chequeo requisitos de columna -> O(nxm)
    for j in range(len(tablero[0])):
        sum_j = sum(fila[j] for fila in tablero)
        if (sum_j != requisitos_col[j]):
            return False

    # Chequeo barcos asignados correctamente -> O(nxm)
    if chequear_barcos(tablero, barcos) == False:
        return False

    return True
```

3.2.2. Detalles del algoritmo

Chequeo barcos diagonales La siguiente función verifica que no haya barcos ubicados de forma diagonal en el tablero:

```python
def chequeo_barcos_diagonales(tablero):
    n = len(tablero)
    m = len(tablero[0])

    for i in range(n):
        for j in range(m):

            if tablero[i][j] == 1:

                # Superior izq -> O(1)
                if i > 0 and j > 0 and tablero[i-1][j-1] == 1:
                    return False

                # Superior der -> O(1)
                if i > 0 and j < m-1 and tablero[i-1][j+1] == 1:
                    return False

                # Inferior izq -> O(1)
                if i < n-1 and j > 0 and tablero[i+1][j-1] == 1:
                    return False

                if i < n-1 and j < m-1 and tablero[i+1][j+1] == 1: -> O(1)
                    return False

    return True
```

Al iterar sobre todas las posiciones del tablero, la complejidad del algoritmo es de $\mathcal{O}(n \times m)$

Chequeo requisitos de fila El siguinte fragmento de código verifica que el tablero cumpla con las demandas de fila pasadas como argumento.

```python
    # Chequeo requisitos de fila -> O(nxm)
    for i in range(len(tablero)):
        if tablero[i].count(1) != requisitos_fil[i]:
            return False
```

Al iterar sobre todas las filas del tablero, y usar la función *count*() (que tiene una complejidad de $\mathcal{O}(m)$, la verificacion tiene una complejidad de $\mathcal{O}(n \times m)$

Chequeo requisitos de columna El siguinte fragmento de código verifica que el tablero cumpla con las demandas de columna pasadas como argumento.

```python
    # Chequeo requisitos de columna -> O(nxm)
    for j in range(len(tablero[0])):
        sum_j = sum(fila[j] for fila in tablero)
        if (sum_j != requisitos_col[j]): # O(1)
            return False
```

Primero itera sobre las columnas del tablero en $\mathcal{O}(m)$ y luego suma los elementos de cada columna iterando sobre las filas del tablero, dando una complejidad total de $\mathcal{O}(n \times m)$

Chequeo de barcos La siguiente función verifica que todos los barcos de la lista pasada como argumento esten ubicados en el tablero:

```python
def chequear_barcos(tablero, barcos):
    n = len(tablero)
    m = len(tablero[0])

    for i in range(n):
        for j in range(m):

            if len(barcos) == 0:
                break

            if tablero[i][j] == 1:
                largo_barco = 1
                tablero[i][j] = 0
                # Barco horizontal
                if j < m-1 and tablero[i][j+1] == 1:
                    while j < m-1:
                        if tablero[i][j+1] == 1:
```

```
                            largo_barco += 1
                            tablero[i][j+1] = 0
                            j += 1
                        else:
                            break

                # Barco vertical
                elif i < n-1 and tablero[i+1][j] == 1:
                    while i < n-1:
                        if tablero[i+1][j] == 1:
                            largo_barco += 1
                            tablero[i+1][j] = 0
                            i += 1
                        else:
                            break

                if barcos.count(largo_barco) > 0:
                    barcos.remove(largo_barco) # O(n)
                else:
                    # No existe un barco con esa longitud
                    return False

    # Quedaron barcos sin asignar al tablero
    if len(barcos) != 0:
        return False

    return True
```

Por cada casillero ocupado, primero se analiza si es un barco vertical o horizontal, para luego buscar el tamaño del barco. Si en la lista de barcos existe un barco con ese tamaño, lo elimina de la lista, sino devuelve False. Al finalizar, si quedaron barcos sin remover de la lista, quiere decir que no todos los barcos fueron ubicados correctamente por lo que devuelve False, de lo contrario devuelve True.

Al iterar sobre todos los casilleros del tablero, la complejidad de la función es de $\mathcal{O}(n \times m)$

3.2.3. Conclusión

La complejidad del validador propuesto es $\mathcal{O}(n \times m)$, lo que es polinomial respecto de los valores de entrada, por lo que podemos afirmar que La Batalla Naval Individual está en NP.

3.3. Demostración NP Completo

A continuación, se demostrará que el problema de la Batalla Naval es un problema NP-Completo, utilizando el problema de Bin-Packing en su versión unaria, como fue recomendado.

Para empezar, primero se debe demostrar que el problema de la Batalla Naval está en NP. Esto fue demostrado en el punto anterior.

Luego, se debe demostrar que el problema de la Batalla Naval está en NP-Completo mediante una reducción de un problema NP-Completo a el problema de la Batalla Naval.

3.3.1. Problema de Bin-Packing en version unaria

Se eligió al problema de Bin-Packing en su version unaria para realizar la reducción al problema de la Batalla Naval. El problema se explica de la siguiente manera:

Dados un conjunto de números S, cada uno expresado en código unario, una cantidad de bins B, expresada en código unario, y la capacidad de cada bin C, expresada en código unario; donde la suma del conjunto de números es igual a C*B, debe decidir si puede cumplirse que los números pueden dividirse en B bins, subconjuntos disjuntos, tal que la suma de elementos de cada bin es exactamente igual a la capacidad C.

3.3.2. Bin-Packing $\leq_P$ Batalla Naval

Debemos demostrar que utilizando una 'caja negra' que resuelve el problema de la Batalla Naval, se puede resolver el problema de Bin-Packing en su versión unaria.

Para esto, debemos transformar polinomialmente la información brindada al problema de Bin-Packing hacia el problema de la Batalla Naval. Se plantea la siguiente transformación:

- Ubicar un barco en una fila del tablero equivale a asignar un numero s_i a un bin.
- El conjunto de números del problema de Bin-Packing serían equivalente al conjunto de barcos en el problema de la Batalla Naval. Es decir, si en una instancia de Bin-Packing tenemos el conjunto {11, 111, 11, 1}, en una instancia de la Batalla Naval tendríramos los barcos {2, 3, 2, 1}.
- Por cada bin, habría una fila en el tablero de la batala naval, y tambien habría una fila más debajo de esta fila. Por lo que la cantidad de bins B (pasada a decimal) sería equivalente a $n/2$, siendo n el numero de filas del tablero.
- La capacidad C de cada bin (pasada a decimal) sería equivalente a la demanda de las filas en el tablero asignadas a cada bin. Las filas intermedias entre bins tendrían 0 de demanda, para evitar barcos adyacentes verticalmente.
- Para evitar barcos adyacentes horizontalmente, la cantidad de columnas m del tablero será igual a $2C - 1$.
- Para las demandas de columnas, se sabe que al ubicar un barco en una celda este cumple 1 de demanda tanto para la fila como para la columna donde esta ubicado. Por lo tanto, el total de la demanda a cumplir por las columnas es $\sum S$. Se debe distribuir uniformemente la demanda a cada columna, con el cálculo de $\frac{\sum S}{2C-1}$ para la demanda de cada columna.
- Si se pueden ubicar los barcos en el tablero cumpliendo con la demanda, que resultan de la transformación planteada, entonces se pueden dividir el conjunto de numeros S en B bins tal que la suma de los elementos de cada bin es exactamente igual a C.

Las anteriores transformaciones se pueden realizar con operaciones polinomiales.

Para demostrar que la reducción es correcta, debemos demostrar que si para cualquier instancia del problema de la Batalla Naval que resulte de la reducción planteada, si existe una solución implica

que para la instancia original del problema de Bin-Packing en su version unaria también existe una solución, y viceversa.

BN → BP Si para la instancia del problema de la Batalla Naval que resulta de la reducción planteada existe una solución, esto implica que:

Debido a que cada fila representa un bin, y cada demanda de fila representa la capacidad de ese bin, si existe una solución quiere decir que todas la demandas de las filas fueron cumplidas, y por lo tanto, la suma del tamaño de los barcos ubicados en cada fila es igual a la demanda de cada fila. Esto se traduce a que la suma de los elementos ubicados en cada bin es igual a la capacidad C, y por lo tanto, existe una solución para el problema de Bin-Packing en su version unaria.

BP → BN Si para una instancia arbitraria del problema de Bin-Packing en su version unaria existe solución, esto implica que:

- El subconjunto de numeros pudo dividirse en los B bines. Esto se traduce a que los barcos pudieron ser ubicados en las filas del tablero.
- La suma de los elementos de cada bin es exactamente igual a la capacidad C. Esto se traduce a que cada fila y columna tienen su demanda cumplida.
- Por la definición de la reducción no se infringen las restricciones de barcos adyacentes ni diagonales.

Y por lo tanto, existe una solución para el problema de la Batalla Naval para la instancia resultante de la reducción planteada.

3.3.3. Conclusión

Hemos demostrado que la reducción es correcta y por ende el problema de Bin-Packing en su version unaria puede ser reducido polinomialmente al problema de la Batalla Naval.

Por lo tanto, podemos afirmar que el problema de la Batalla Naval es un problema NP-Completo.

3.4. Backtracking

En la siguiente sección se resolverá la versión de optimización de la Batalla Naval Individual mediante un algoritmo de Backtracking. Este algoritmo explora todas las posibilidades de solucion, realizando podas para evitar calcular soluciones que ya no son válidas o mejores que la solucion actual, y devuelve la solución óptima.

3.4.1. Aclaraciones previas

Contrario a puntos anteriores, los barcos se representarán en el tablero con un número de indice. De esta manera se indica no sólo la posición de los barcos en el tablero, sino qué barco esta ubicado.

3.4.2. Detalles del algoritmo

A continuación se mostrará el código del algoritmo planteado:

```
def batalla_naval_bt(tablero, barcos, demanda_fila, demanda_columna, mejor_demanda_inc, demanda_inc, indice, mejor_tablero):
    if mejor_demanda_inc == 0:
        return mejor_tablero, mejor_demanda_inc

    if not barcos:
        demanda_inc = calcular_demanda_incumplida(demanda_fila, demanda_columna)
        if demanda_inc < mejor_demanda_inc:
            mejor_demanda_inc = demanda_inc
            mejor_tablero = copy.deepcopy(tablero)
        return mejor_tablero, mejor_demanda_inc

    barcos_filtrados = filtrar_barcos_por_demanda(barcos, demanda_fila, demanda_columna)
    if not barcos_filtrados:
        return mejor_tablero, mejor_demanda_inc

    # Si con los barcos que me quedan no puedo mejorar la mejor demanda incumplida, corto la rama
    if calcular_demanda_incumplida(demanda_fila, demanda_columna) - sum(barcos_filtrados)*2 >= mejor_demanda_inc:
        return mejor_tablero, mejor_demanda_inc

    posiciones = calcular_posibles_posiciones(tablero, len(tablero), len(tablero[0]), barcos_filtrados[0], demanda_fila, demanda_columna)

    mejor_tablero_actual = mejor_tablero

    for posicion in posiciones:

        if calcular_demanda_incumplida(demanda_fila, demanda_columna) - sum(barcos_filtrados)*2 >= mejor_demanda_inc:
            return mejor_tablero, mejor_demanda_inc

        nuevo_tablero = marcar_barco_en_tablero(tablero, indice, barcos_filtrados[0], posicion)

        nueva_demanda_fila, nueva_demanda_columna = actualizar_demandas(demanda_fila, demanda_columna, barcos_filtrados[0], posicion)

        # Si el barco actual no se puede ubicar, todos los barcos con igual tamano tampoco, obtengo el primero distinto
        if posicion == None:
            i_prox_barco =  obtener_indice_prox_barco(barcos_filtrados)
        else:
            i_prox_barco = 1

        nuevo_tablero, nuevo_demanda_inc = batalla_naval_bt(nuevo_tablero, barcos_filtrados[i_prox_barco:], nueva_demanda_fila, nueva_demanda_columna, mejor_demanda_inc, demanda_inc, indice+1, mejor_tablero_actual)
```

```
40
41          if nuevo_demanda_inc < mejor_demanda_inc:
42              mejor_demanda_inc = nuevo_demanda_inc
43              mejor_tablero = nuevo_tablero
44
45      return mejor_tablero, mejor_demanda_inc
```

El algoritmo funciona de la siguiente manera:

1. La condición de corte de la función recursiva es al no quedar barcos por ubicar, se calcula la demanda incumplida de la solución actual y si es menor a la mejor, se actualiza.
2. La primera poda se ve al filtrar los barcos por demanda. Si hay barcos cuyo tamaño es mayor a la máxima demanda permitida por filas o columnas, se eliminan de la solución.
3. La segunda poda se ve al cortar la rama de solución si es que con los barcos que me quedan por ubicar no puedo mejorar la demada incumplida por la mejor solución.
4. Luego, se calculan las posibles posiciones para el barco actual. Para calcular esto se tienen en cuenta los barcos ya ubicados en el tablero, las demandas de filas y columnas actuales de la solución, y las restricciones de barcos contiguos y diagonales.
5. Se itera sobre las posiciones calculadas, se marca el barco en el tablero en la posicion indicada y se actualizan las demandas.
6. Previo al llamado recursivo, se realiza otra poda. Si no hay posiciones válidas para ubicar el barco actual (es decir posicion = None), quiere decir que ningún barco del mismo tamaño va a poder ser ubicado. Para eso se calcula el índice del primer barco con tamaño distinto al actual y se llama a la función recursiva con ese barco.
7. Al volver del llamado recursivo, se analiza si la solución devuelta es mejor que la mejor solución actual, y se actualiza, para luego devolver la mejor solución.

3.4.3. Resultados

A continuación, se mostrarán los tiempos de ejecución del algoritmo de backtracking.

Generación de sets de datos Para medir con mas precisión los tiempos de ejecución, se generaron sets de datos propios con la siguiente función:

```
1  # Genera n sets de datos de tamano incremental para el problema de la Batalla Naval
       .
2  # Si se especifica una semilla, se utilizara para generar los datos.
3  def generar_set_datos_batalla_naval(semilla, n):
4      if semilla != None:
5          seed(semilla)
6
7      set_datos = []
8
9      for i in range(1, n+1):
10         n = i * 3
11         m = i * 3
12         k = i * 2
13         barcos = [randint(1, min(i * 3, 16)) for _ in range(k)]
14         demandas_filas = [randint(0, min(i * 3, 15)) for _ in range(n)]
15         demandas_columnas = [randint(0, min(i * 3, 15)) for _ in range(m)]
16
17         set_datos.append((barcos, demandas_filas, demandas_columnas))
18
19     return set_datos
```

De esta forma, se generan sets de datos que van incrementando iterativamente el tamaño del tablero y la cantidad de barcos, aplicandole un límite al tamaño de barcos y de demandas para que no sean excesivas.

N° Set	1	2	3	4	5	6	7	8	9	10
Número de Barcos	2	4	6	8	10	12	14	16	18	20
Tamaño del Tablero	3x3	6x6	9x9	12x12	15x15	18x18	21x21	24x24	27x27	30x30

Cuadro 1: Sets de datos generados.

Utilizando el método mencionado anteriormente, se generaron 10 sets de datos como se muestran a continuación:

Mediciones

Los resultados de las mediciones de tiempo obtenidas se muestran a continuación:

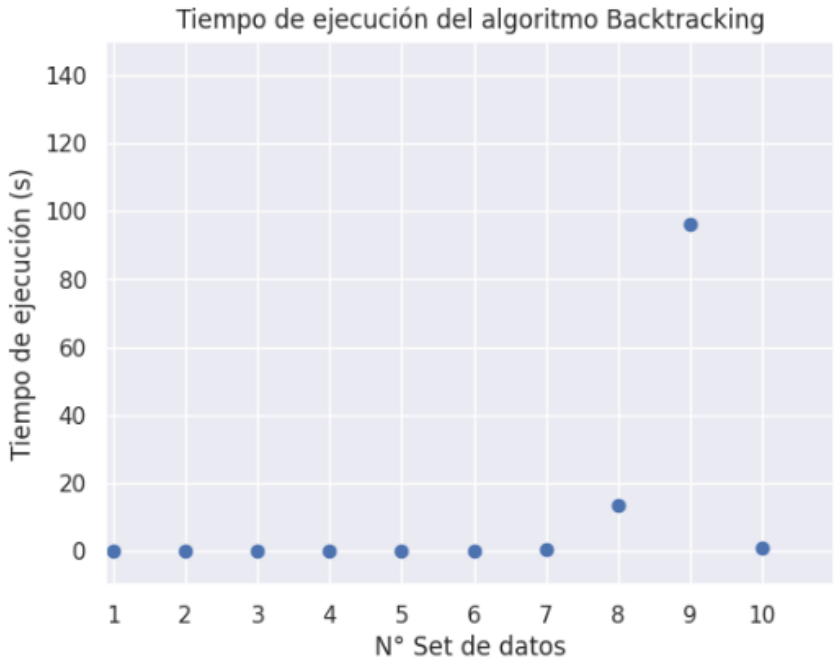

Figura 9: Mediciones en segundos del tiempo de ejecución del algoritmo Backtracking

3.4.4. Anánilis de los resultados

Analizando el gráfico mostrado de los tiempos de ejecución del algoritmo de backtracking, podemos ver claramente como el algoritmo funciona casi instantáneamente para sets de datos de tamaño pequeño, pero a medida que se usan sets de datos mas grandes, el algoritmo tarda considerablemente más. Esto demuestra la complejidad teórica exponencial de los algoritmos de backtracking.
También podemos mencionar que hay ciertas configuraciones de barcos y demandas que hacen que el algoritmo tarde menos, como vemos en el set de datos 10, que por más de ser de mayor tamaño que los sets de datos 8 y 9, el tiempo de ejecución mucho más rápido.

3.5. Programación Lineal

En la siguiente sección se resolverá la versión de optimización de la Batalla Naval Individual mediante un algoritmo de Programación Lineal Entera. Este algoritmo define una serie de ecuaciones lineales, que representan variables del problema y sus restricciones, y luego busca los valores de las variables definidas que optimizen una función objetivo. Para resolverlo se utilizó la librería de Python PULP.

3.5.1. Modelo planteado

Variables del modelo Para resolver el problema de optimización de la Batalla Naval Individual usando Programación Lineal Entera, se definieron las siguientes variables:

- $x_{i,j,b,o}$: Indica si el barco ***b*** tiene como posición inicial al casillero (***i,j***) del tablero y con orientación ***o***. La orientación representa si un barco esta ubicado vertical u horizontalmente.
- u_i: Representa la demanda incumplida de la fila i. Es un valor entero.
- v_j: Representa la demanda incumplida de la columna j. Es un valor entero.

Se definen las siguientes restricciones para las variables:

Restricciones del modelo Por cada fila i del tablero, la suma entre las celdas ocupadas en esa fila y la demanda incumplida de la fila (u_i), debe ser igual a la demanda de la fila i:

$\sum_{j=0}^{m} \sum_{b=0}^{k} x_{i,j,b,0} * barcos[b] + \sum_{j=1}^{m} \sum_{b=0}^{k} \sum_{l=0}^{barcos[b]} x_{i-l,j,b,1} + u_i = demanda\_filas[i], \forall i$

Por cada columna j del tablero, la suma entre las celdas ocupadas en esa columna y la demanda incumplida de la columna (v_j), debe ser igual a la demanda de la columna j:

$\sum_{i=0}^{n} \sum_{b=0}^{k} x_{i,j,b,1} * barcos[b] + \sum_{i=0}^{n} \sum_{b=0}^{k} \sum_{l=0}^{barcos[b]} x_{i,j-l,b,0} + v_j = demanda\_columnas[j], \forall j$

Cada barco debe tener una única posicion y orientacion inicial:

$\sum_{i=0}^{n} \sum_{j=0}^{m} \sum_{o=0}^{1} x_{i,j,b,o} \leq 1, \quad \forall b$

No puede haber dos barcos con la misma posicion en el tablero:

$\sum_{b=0}^{k} \sum_{o=0}^{1} x_{i,j,b,o} \leq 1, \quad \forall i, j$

Ningun barco puede estar ubicado de forma contigua o diagonal con otro barco:

Caso horizontal ($o = 0$):

$$\forall (i,j,b) \text{ con } j + \text{barcos}[b] \leq m : \quad \sum_{\substack{(ni,nj,b',o') \\ b' \neq b}} x[ni, nj, b', o'] \leq 1 - x[i, j, b, 0],$$

donde:

$$(ni, nj) = (i + \Delta_i, j + l + \Delta_j), \quad \Delta_i, \Delta_j \in \{-1, 0, 1\}, \quad l \in \{0, \ldots, \text{barcos}[b] - 1\}.$$

Caso vertical ($o = 1$):

$$\forall (i,j,b) \text{ con } i + \text{barcos}[b] \leq n : \quad \sum_{\substack{(ni,nj,b',o') \\ b' \neq b}} x[ni, nj, b', o'] \leq 1 - x[i, j, b, 1],$$

donde:

$$(ni, nj) = (i + l + \Delta_i, j + \Delta_j), \quad \Delta_i, \Delta_j \in \{-1, 0, 1\}, \quad l \in \{0, \ldots, \text{barcos}[b] - 1\}.$$

Por último, como función objetivo se planteó la minimización de las demandas incumplidas:

$\sum_{i=0}^{n} u_i + \sum_{j=0}^{m} v_j$

3.5.2. Detalles del algoritmo

A continuación se mostrará el código del algoritmo de programación lineal entera planteado:

```
def batalla_naval_individual_pl(n, m, k, barcos, demandas_filas, demandas_columnas):

    # Crear el problema
    problema = LpProblem("Minimizar_demanda_incumplida", LpMinimize)

    # Variables de decision
    # x -> Barco b en la posicion (i, j) y orientacion o
    # u -> Demanda incumplida en fila i
    # v -> Demanda incumplida en columna j
    x = LpVariable.dicts("x", ((i, j, b, o) for i in range(n) for j in range(m) for b in range(k) for o in [0, 1]
        if es_posicion_valida(barcos, i, j, b, o, demandas_filas, demandas_columnas, n, m)), cat=LpBinary)

    u = LpVariable.dicts("u", (i for i in range(n)), lowBound=0, cat=LpInteger)
    v = LpVariable.dicts("v", (j for j in range(m)), lowBound=0, cat=LpInteger)

    # Funcion objetivo: Minimizar demanda incumplida
    problema += (
        lpSum(u[i] for i in range(n)) +
        lpSum(v[j] for j in range(m))
    ), "Minimizar_Demanda_Incumplida"

    # Restricciones de demanda en filas
    for i in range(n):
        problema += (
            lpSum(
                x[i, j, b, 0] * barcos[b] for j in range(m) for b in range(k) if (i,j,b,0) in x if j + barcos[b] <= m
            ) +
            lpSum(
                x[i - l, j, b, 1] for j in range(m) for b in range(k) for l in range(barcos[b]) if (i-l,j,b,1) in x if 0 <= i - l < n
            ) +
            u[i] == demandas_filas[i]
        )

    # Restricciones de demanda en columnas
    for j in range(m):
        problema += (
            lpSum(
                x[i, j, b, 1] * barcos[b] for i in range(n) for b in range(k) if (i,j,b,1) in x if i + barcos[b] <= n
            ) +
            lpSum(
                x[i, j - l, b, 0] for i in range(n) for b in range(k) for l in range(barcos[b]) if (i,j-l,b,0) in x if 0 <= j - l < m
            ) +
            v[j] == demandas_columnas[j]
        )


    # Restricciones de unicidad de los barcos
    for b in range(k):
        problema += (
            lpSum(x[i, j, b, o] for i in range(n) for j in range(m) for o in [0, 1] if (i,j,b,o) in x) <= 1
        )

    # Restricciones de no solapamiento
    for i in range(n):
        for j in range(m):
            problema += (
                lpSum(
                    x[i, j, b, o] for b in range(k) for o in [0, 1] if (i,j,b,o) in x
```

```
                ) <= 1
            ), f"No_Solapamiento_{i}_{j}"

    # Restricciones para evitar barcos contiguos y diagonales
    for (i, j, b, o) in x:
        if o == 0 and j + barcos[b] <= m:  # Barco horizontal
            for l in range(barcos[b]):
                for di, dj in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1)
    , (1, 0), (1, 1)]:
                    ni, nj = i + di, j + l + dj
                    if 0 <= ni < n and 0 <= nj < m:
                        problema += (
                            lpSum(x[ni, nj, b2, o2] for (ni2, nj2, b2, o2) in x if
    ni2 == ni and nj2 == nj and b2 != b) <= 1 - x[i, j, b, o]
                        )
        elif o == 1 and i + barcos[b] <= n:  # Barco vertical
            for l in range(barcos[b]):
                for di, dj in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1)
    , (1, 0), (1, 1)]:
                    ni, nj = i + l + di, j + dj
                    if 0 <= ni < n and 0 <= nj < m:
                        problema += (
                            lpSum(x[ni, nj, b2, o2] for (ni2, nj2, b2, o2) in x if
    ni2 == ni and nj2 == nj and b2 != b) <= 1 - x[i, j, b, o]
                        )


    # Resolver el modelo
    problema.solve(PULP_CBC_CMD())
```

De esta forma, se crea el modelo y se resuelve usando el solver de PULP, asignando las posiciones iniciales de los barcos de manera que se minimize la demanda incumplida total.

3.5.3. Resultados

Para mostrar la eficiencia del algoritmo de Programación Lineal Entera planteado, se ejecutará sobre el mismo set de datos usado para el algoritmo de Backtracking, y se compararán las mediciones de tiempos obtenidos.

Mediciones Se muestran, en segundos, las mediciones temporales de ejecución de los algoritmos de Backtracking y de Programación Lineal Entera, ajustados a 5 cifras significativas. Se limitó hasta el set n°8 debido a la complejidad del algoritmo.

N° Set	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Tiempo BT	0.0002	0.0002	0.0023	0.2467	0.0628	0.0399	0.4198	13.610
Tiempo PL	0.02	0.05	3.79	2273.5	1.96	41.26	125.28	> 5000

Cuadro 2: Tabla comparativa de tiempos de ejecución para PL y BT.

3.5.4. Análisis de los resultados

Se ve claramente que el algoritmo de programación lineal entera es significativamente mas lento que el de backtracking. Para sets de datos de pequeño y mediano tamaño (salvo el set 4) el algoritmo entrega la solución óptima en tiempos razonables, pero para sets grandes el algoritmo tarda un tiempo que hace que no valga la pena elegir este algoritmo por sobre el de backtracking.

3.6. Algoritmo de Aproximación

En la siguiente sección se analizara el algoritmo de aproximación planteado por John Jellicoe para resolver el problema de la Batalla Naval.

3.6.1. Algoritmo propuesto

El algoritmo propuesto por el almirante es el siguiente:

'Ir a fila/columna de mayor demanda, y ubicar el barco de mayor longitud en dicha fila/columna en algún lugar válido. Si el barco de mayor longitud es más largo que dicha demanda, simplemente saltearlo y seguir con el siguiente. Volver a aplicar hasta que no queden más barcos o no haya más demandas a cumplir.'

Para eso, planteamos el siguiente código:

```python
def batalla_naval_individual_aprox(tablero, demanda_filas, demanda_columnas, barcos):

    indice_barco = 1
    flag_marcado = True
    n = len(tablero)
    m = len(tablero[0])

    # Mientras haya demanda por cumplir y barcos por ubicar
    while queda_demanda_por_cumplir(demanda_filas, demanda_columnas) and len(barcos) > 0:
        if not flag_marcado:
            break

        flag_marcado = False

        # Obtengo el indice de la fila/columna con mayor demanda y su tipo (fila/columna)
        indice_max_demanda, tipo = obtener_max_demanda(demanda_filas, demanda_columnas)

        for barco in barcos:

            # Obtengo una posicion valida para ubicar el barco de mayor longitud
            posicion = posicion_valida(tablero, barco, n, m, demanda_filas, demanda_columnas, indice_max_demanda, tipo)
            if posicion is None:
                continue

            if tipo == "fila":
                if demanda_filas[indice_max_demanda] >= barco:
                    marcar_barco_y_actualizar_demandas(tablero, barco, posicion, demanda_filas, demanda_columnas)
                    barcos.remove(barco)
                    indice_barco += 1
                    flag_marcado = True
                    break

            else:
                if barco <= len(tablero) and demanda_columnas[indice_max_demanda] >= barco:
                    marcar_barco_y_actualizar_demandas(tablero, barco, posicion, demanda_filas, demanda_columnas)
                    barcos.remove(barco)
                    indice_barco += 1
                    flag_marcado = True
                    break

            if flag_marcado:
                break

    return tablero, sum(demanda_filas) + sum(demanda_columnas)
```

Los barcos fueron ordenados de mayor a menor antes del llamado al algoritmo.

3.6.2. Análisis de la complejidad

A continuación analizaremos los fragmentos del algoritmo planteado y su complejidad, para poder analizar la complejidad de la totalidad del algoritmo de aproximación.

Preparación de la solución Antes de llamar al algoritmo, se realizan algunas operaciones como:

- Construir el tablero vacio: $O(nxm)$
- Ordenar los barcos de mayor a menor: $O(klog(k))$, siendo k la cantidad de barcos.

Lógica principal La función principal del algoritmo itera mientras haya demandas o barcos $O(k)$, y en cada iteracion se realizan las siguientes operaciones:

- Obtener la máxima demanda, con una complejidad de $O(n + m)$
- Por cada barco (k barcos):
  - Analizar si hay una posición válida $O(max(n, m) * b)$
  - Marcar el barco en el tablero y actualizar las demandas $O(b)$

Por lo que la complejidad de la función principal es $O(k * [n + m + k * max(n, m) * b])$

Complejidad total Por lo tanto, la complejidad total del algoritmo de aproximación, teniendo en cuenta tanto la preparación de la solución como el bucle principal es:

$O(n * m + k * log(k) + k * [n + m + k * max(n, m) * b)$

Comparándolo con las soluciones por Backtracking o Programación Lineal Entera, que ambos tienen complejidad exponencial, parece reducir significativamente los tiempos de ejecución, ya que como peor caso no parece superar la complejidad cuadrática.

3.6.3. Mediciones

Los resultados de los tiempos de ejecución obtenidos utilizando el algoritmo de aproximación se muestran a continuación, utilizando el mismo set de datos que para el algoritmo de backtracking:

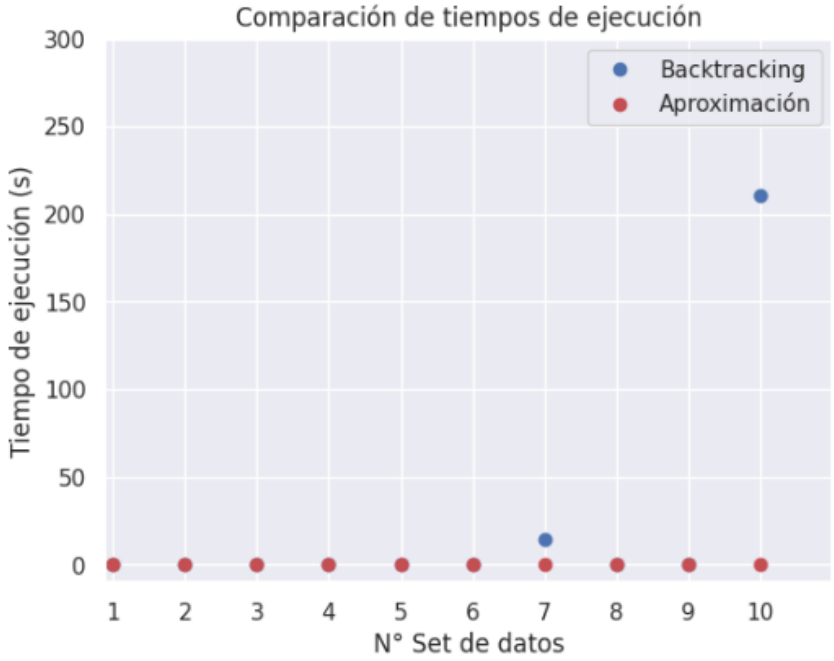

Figura 10: Gráfico comparativo de los tiempos de ejecución de Backtracking y Aproximación

Se ve claramente como el algoritmo de aproximación funciona de manera casi instantánea para todos los sets de datos, mientras que el algoritmo de backtracking sufre cuando se le pasan sets de datos de gran tamaño.

3.6.4. Análisis de la aproximación

Sets de datos brindados por la cátedra A continuación se realizará una evaluación de qué tan buena es la aproximación realizada, comparando la solución aproximada con la solución óptima obtenida de los algoritmos de backtracking o programación lineal, utilizando los casos de pruebas brindados por la cátedra.

Para eso, definimos:

- I como una instancia cualquiera del problema de la Batalla Naval.
- $z(I)$ como la solución óptima para la instancia I.
- $A(I)$ como la solución aproximada para la instancia I.
- $r(A) \leq \frac{A(I)}{z(I)}$ una cota del peor caso que puede resultar de aplicar el algoritmo. Esto nos dará una idea de qué tan bueno es.

Podemos ver que la cota máxima de error es de 0.5643, es decir, que usando el algoritmo aproximado podemos obtener como peor caso, casi la mitad de la demanda cumplida óptima.

Sets de datos generados aleatoriamente

Utilizando la función mostrada para el ejercicio de Backtracking, generaremos 10 sets de datos como se muestran a continuación, y se compararán los resultados para los algoritmos de backtracking y de aproximación:

Caso de Prueba (I)	$z(I)$	$A(I)$	$r(A)$
3_3_2.txt	4	4	1
5_5_6.txt	12	12	1
8_7_10.txt	26	26	1
10_3_3.txt	6	6	1
10_10_10.txt	40	32	0.8
12_12_21.txt	46	40	0.8695
15_10_15.txt	40	38	0.95
20_20_20.txt	104	104	1
20_25_30.txt	172	152	0.8837
30_25_25.txt	202	114	0.5643

Cuadro 3: Tabla comparativa entre solución óptima y solución aproximada con casos de prueba.

N° Set	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Número de Barcos	2	4	6	8	10	12	14	16	18	20
Tamaño del Tablero	3x3	6x6	9x9	12x12	15x15	18x18	21x21	24x24	27x27	30x30

Cuadro 4: Sets de datos generados.

Set de datos (I)	$z(I)$	$A(I)$	$r(A)$
1	2	2	1
2	14	10	0.7142
3	30	30	1
4	76	24	0.3157
5	30	30	1
6	56	56	1
7	160	102	0.6375
8	226	204	0.9026
9	208	208	1
10	234	234	1

Cuadro 5: Tabla comparativa entre solución óptima y solución aproximada con sets de datos.

En este caso, podemos ver que la cota máxima de error es de 0.3157, es decir, que usando el algoritmo aproximado podemos obtener como peor caso, menos de la mitad de la demanda cumplida óptima.

3.6.5. Conclusión

Dado la cota calculada podemos decir que el algoritmo de aproximación planteado por John Jellicoe para resolver el problema de la Batalla Naval no es muy bueno. Vimos que en los sets generados aleatoriamente la relación entre la solución óptima y la aproximada, como peor caso, difiere en un factor de aproximadamente 3/10.

3.7. Conclusiones

En este último trabajo práctico se analizaron diferentes algoritmos para resolver el problema de la Batalla Naval, y se demostró que es un problema NP-Completo.

A continuación, haremos un análisis mas detallado de los resultados observados

3.7.1. Características de los algoritmos

Para resolver el problema de la Batalla Naval se utilizaron 3 algoritmos, cada uno con sus características propias y sus ventajas y desventajas:

- **Backtracking**: Devuelve siempre la solución óptima explorando todas las posibles configuraciones de barcos en el tablero brindado, optimizando la búsqueda con podas. Es un algoritmo que funciona muy rápido para sets de datos de pequeño y mediano tamaño, pero debido a su complejidad exponencial sufre con sets de datos de gran tamaño.
- **Programación Lineal Entera**: También devuelve siempre la solución óptima, modelando al problema como un sistema de ecuaciones lineales. Es un algoritmo que funciona rápido para sets de datos de pequeño tamaño, pero con algunas determinadas configuraciones, y sets de tamaño grande se hace inutilizable por el tiempo de ejecución. También esto se debe a su naturaleza exponencial.
- **Algoritmo de Aproximación**: Este algoritmo no devuelve la solución óptima y funciona con una rapidez significativamente mejor a las demás opciones. Dadas las mediciones, concluímos que el algoritmo planteado no es una buena aproximación, pero si se quiere tener una respuesta rápida para sets de datos inmanejables por los demas algoritmos, es una buena opción.

3.7.2. Observaciones

Más allá de el tamaño de los sets de datos utilizados en cada algoritmo, hemos notado que hay ciertas configuraciones de barcos y demandas que hacen que los algoritmos exactos (backtracking y progrmación lineal) funcionen con mayor lentitud. Esto se puede ver en las mediciones en el set n° 8 y 9 del algoritmo de backtracking, o en el set n° 4 de programación lineal entera.

Creemos que se puede deber a que ciertas configuraciones no encuentrar una solución lo suficientemente buena de manera rápida, por lo que tardan mucho en encontrar el óptimo.

3.7.3. Conclusión Final

Hemos aboradado el problema de la Batalla Naval desde distintas perspectivas, pudiendo profundizar en las características de cada enfoque, como sus ventajas y limitaciones. De esta forma, finalizamos el trabajo con un mejor entendimiento de qué analizar a la hora de tomar una decisión sobre que tipo de algoritmo utilizar.

3.8. Anexo Correcciones

En esta sección, se desarrollarán las correcciones a los diversos puntos luego de la primera entrega. ——

3.8.1. Mejorar la explicacion del modelo de programación lineal

A continuación, se detallará nuevamente el modelo de programación lineal entera planteado junto con sus restricciones, agregando un mayor grado de detalle y explicación.

Variables del modelo Para resolver el problema de optimización de la Batalla Naval Individual usando Programación Lineal Entera, se definieron las siguientes variables:

- $x_{i,j,b,o}$: Indica si el barco $\boldsymbol{b}$ tiene como posición inicial al casillero $(\boldsymbol{i,j})$ del tablero y con orientación $\boldsymbol{o}$. La orientación representa si un barco esta ubicado vertical u horizontalmente.
  - Si la variabe $x_{i,j,b,0}$ tiene valor 1, quiere decir que el barco b está ubicado en la posicion (i, j) con orientación horizontal.
  - Si la variabe $x_{i,j,b,1}$ tiene valor 1, quiere decir que el barco b está ubicado en la posicion (i, j) con orientación vertical.
- u_i: Representa la demanda incumplida de la fila i. Es un valor entero.
- v_j: Representa la demanda incumplida de la columna j. Es un valor entero.
- $barcos$: Una lista de tamaño k donde el barco i tiene b_i de largo.
- $demanda\_filas$: Una lista de tamaño n donde cada fila i tiene que cumplir con d_i de demanda.
- $demanda\_columnas$: Una lista de tamaño m donde cada columna j tiene que cumplir con d_j de demanda.

Se definen las siguientes restricciones para las variables:

Restricciones del modelo **Por cada fila $\boldsymbol{i}$ del tablero, la suma entre las celdas ocupadas en esa fila y la demanda incumplida de la fila (u_i), debe ser igual a la demanda de la fila $\boldsymbol{i}$**:

$$\sum_{j=0}^{m}\sum_{b=0}^{k} x_{i,j,b,0} * barcos[b] + \sum_{j=1}^{m}\sum_{b=0}^{k}\sum_{l=0}^{barcos[b]} x_{i-l,j,b,1} + u_i = demanda\_filas[i], \forall i$$

- Iteramos sobre todas las columnas de la fila, y por cada barco ubicado horizontalmente sumamos su largo, y por cada barco ubicado verticalmente, sumamos 1. Esto sumado a la demanda incumplida u_i de la fila debe ser igual a la demanda d_i de la fila.

Por cada columna $\boldsymbol{j}$ del tablero, la suma entre las celdas ocupadas en esa columna y la demanda incumplida de la columna (v_j), debe ser igual a la demanda de la columna $\boldsymbol{j}$:

$$\sum_{i=0}^{n}\sum_{b=0}^{k} x_{i,j,b,1} * barcos[b] + \sum_{i=0}^{n}\sum_{b=0}^{k}\sum_{l=0}^{barcos[b]} x_{i,j-l,b,0} + v_j = demanda\_columnas[j], \forall j$$

Cada barco debe tener una única posicion y orientacion inicial:

$$\sum_{i=0}^{n}\sum_{j=0}^{m}\sum_{o=0}^{1} x_{i,j,b,o} \leq 1, \quad \forall b$$

- No podemos permitir que un barco esté ubicado en dos posiciones del tablero, por lo que iteramos sobre todas las posiciones y orientaciones posibles, y restringimos la variable de decisión $x_{i,j,b,o}$ a ≤ 1 (puede no estar ubicado también). Hacemos esto para todos los barcos.

- Por ejemplo, no puede pasar que tanto $x_{1,1,1,0}$ como $x_{1,3,1,0}$ tengan valor 1, ya que estaríamos diciendo que el barco 1 esta ubicado tanto en la posición $(1,1)$ como en la $(1,3)$.

No puede haber dos barcos con la misma posicion en el tablero:
$\sum_{b=0}^{k} \sum_{o=0}^{1} x_{i,j,b,o} \leq 1, \quad \forall i,j$

- No podemos permitir que el modelo elija ubicar un mismo barco en la misma posicion pero con las dos orientaciones posibles. Con esta restricción evitamos esto para cada posible posicion del tablero.
- Por ejemplo, no puede pasar que tanto $x_{2,2,1,0}$ como $x_{2,2,1,1}$ tengan valor 1. De lo contrario, estaríamos diciendo que el barco 1 esta ubicado en la posición (2,2) vertical y horizontalmente.

Ningun barco puede estar ubicado de forma contigua o diagonal con otro barco:

Caso horizontal $(o = 0)$:

$$\forall (i,j,b) \text{ con } j + \text{barcos}[b] \leq m : \sum_{\substack{(ni,nj,b',o') \\ b' \neq b}} x[ni,nj,b',o'] \leq 1 - x[i,j,b,0],$$

donde:

$$(ni,nj) = (i + \Delta_i, j + l + \Delta_j), \quad \Delta_i, \Delta_j \in \{-1,0,1\}, \quad l \in \{0,\ldots,\text{barcos}[b]-1\}.$$

- Si el barco b está ubicado en la posición (i,j). iteramos sobre todas las celdas adyacentes a las posiciones que ocupe el barco (al estar en el caso horizontal, sería la posicion inicial y las posiciones adyacentes hacia la derecha segun el tamaño del barco) y nos aseguramos que no haya otro barco ubicado en esas celdas.

Caso vertical $(o = 1)$:

$$\forall (i,j,b) \text{ con } i + \text{barcos}[b] \leq n : \sum_{\substack{(ni,nj,b',o') \\ b' \neq b}} x[ni,nj,b',o'] \leq 1 - x[i,j,b,1],$$

donde:

$$(ni,nj) = (i + l + \Delta_i, j + \Delta_j), \quad \Delta_i, \Delta_j \in \{-1,0,1\}, \quad l \in \{0,\ldots,\text{barcos}[b]-1\}.$$

- Si el barco b está ubicado en la posición (i,j). iteramos sobre todas las celdas adyacentes a las posiciones que ocupe el barco (al estar en el caso vertical, sería la posicion inicial y las posiciones adyacentes hacia abajo segun el tamaño del barco) y nos aseguramos que no haya otro barco ubicado en esas celdas.

Función objetivo

Por último, como función objetivo se planteó la minimización de las demandas incumplidas:
$\sum_{i=0}^{n} u_i + \sum_{j=0}^{m} v_j$

3.8.2. Detalles del algoritmo de Backtracking

Se nos solicitó brindar más detalles sobre la función de Backtracking, específicamente de la función *calcular_posibles_posiciones* y la configuración inicial de las variables.

Configuración inicial y detalle de las variables A continuación, detallaremos las variables que forman parte de nuestro algoritmo de Backtracking:

```
def batalla_naval_bt(tablero, barcos, demanda_fila, demanda_columna,
mejor_demanda_inc, demanda_inc, indice, mejor_tablero):
```

- **tablero**: Variable que almacena una matriz $n \times m$ donde se ubican los barcos. En las celdas de la matriz donde, por ejemplo, se ubique el barco 1, se almacenará un 1. Se actualiza en cada llamado.
- **barcos**: Lista de k barcos donde el barco i tiene un tamaño b_i.
- **demanda_fila**: Lista de tamaño n donde la fila i debe cumplir con df_i de demanda. Al ubicar un barco en el tablero, se actualizan las demandas de las filas implicadas.
- **demanda_columna**: Lista de tamaño m donde la columna j debe cumplir con dc_i de demanda. Al ubicar un barco en el tablero, se actualizan las demandas de las columnas implicadas.
- **mejor_demanda_incumplida**: Variable que almacena el mejor resultado obtenido por el algoritmo.
- **demanda_incumplida**: Variable que almacena el resultado de la solución actual, es decir, cuánta demanda queda por cumplir en el tablero actual.
- **indice**: Variable usada para marcar el barco correcto en el tablero.
- **mejor_tablero**: Variable que almacena el tablero final de la mejor solución.

La configuración inicial de nuestro algoritmo es la siguiente:

```
1  barcos = sacar_barcos_muy_grandes(len(tablero), len(tablero[0]), barcos,
   demanda_fila, demanda_columna)
2  barcos.sort(reverse=True)
3  mejor_tablero, mejor_demanda_inc = batalla_naval_bt(tablero, barcos,
   demanda_fila, demanda_columna, float('inf'), 0, 1, tablero)
```

Como vemos, primero se filtran los barcos que por su tamaño es imposible ubicarlos, luego se ordenan de mayor a menor y por último se llama al algoritmo de backtracking como se muestra en el código.

Detalles de la función calcular_posibles_posiciones

La función, como indica el nombre, calcula las posibles posiciones del tablero (junto con la orientación) en las que puede ser ubicado el barco actual. Para esto se tienen en cuenta los barcos ya ubicados en el tablero, las demandas de filas y columnas actuales de la solución, y las restricciones de barcos contiguos y diagonales.

A continuación, se presenta el código que ejecuta la función:

```
1  def calcular_posibles_posiciones(tablero, n, m, tamano_barco, demanda_fila,
   demanda_columna):
2      posiciones = []
3
4      for i in range(n):
5          if demanda_fila[i] < tamano_barco:
6              continue
7
8          for j in range(m - tamano_barco + 1):
9              if demanda_columna[j] < 1:
10                 continue
11             posicion = ((i, j), 'horizontal')
```

```
            if not supera_demanda_permitida(tablero, posicion, tamano_barco, demanda_fila, demanda_columna) and es_posicion_valida(tablero, tamano_barco, posicion, n, m):
                posiciones.append(posicion)

    if tamano_barco == 1:
        posiciones.append(None)
        return posiciones

    for i in range(n - tamano_barco + 1):
        if demanda_fila[i] < 1:
            continue
        for j in range(m):
            if demanda_columna[j] < tamano_barco:
                continue
            posicion = ((i, j), 'vertical')
            if not supera_demanda_permitida(tablero, posicion, tamano_barco, demanda_fila, demanda_columna) and es_posicion_valida(tablero, tamano_barco, posicion, n, m):
                posiciones.append(posicion)

    posiciones.append(None)
    return posiciones
```

La función sigue los siguientes pasos:

- Primero busca las posibles posiciones horizontales en que se puede posicionar el barco.
  - Por cada fila, primero analiza si puede ubicar el barco en base a su tamaño y a la demanda de la fila. En caso de no poder, saltea la fila.
  - En caso de poder, itera sobre las columnas de esa fila, y analiza si puede ubicar el barco en cada una de ellas en base a el tamaño del barco, la demanda faltante y si es una posición válida (chequea que no haya barcos adyacentes ni diagonales).
  - Por cada posición válida, la guarda en una lista junto con su orientación.
- Luego, repite el proceso para posiciones verticales.

De esta forma, nos evitamos probar con todas las posibles posiciones del tablero a la hora de ubicar un barco, ya que podemos descartar de antemano las que ya sabemos que son posiciones inválidas.

3.8.3. Corrección complejidad de certificador polinomial

Tuvimos un error a la hora de calcular la complejidad del algoritmo presentado como certificador polinomial.

A continuación, se mostrará nuevamente el código genérico del certificador con su complejidad correcta, y luego se detallará la complejidad de la función que calculamos incorrectamente.

Algoritmo certificador polinomial

```
def certificador_eficiente(tablero, barcos, requisitos_fil, requisitos_col):

    # Chequeo barcos diagonales -> O(nxm)
    if chequeo_barcos_diagonales(tablero) == False:
        return False

    # Chequeo requisitos de fila -> O(nxm)
    for i in range(len(tablero)):
        if tablero[i].count(1) != requisitos_fil[i]:
            return False

    # Chequeo requisitos de columna -> O(nxm)
    for j in range(len(tablero[0])):
        sum_j = sum(fila[j] for fila in tablero)
```

```
        if (sum_j != requisitos_col[j]):
            return False

    # Chequeo barcos asignados correctamente -> O(n x m x sum(barcos))
    if chequear_barcos(tablero, barcos) == False:
        return False

    return True
```

Detalles del algoritmo Como mencionamos, habíamos calculado mal la complejidad de la función *chequear_barcos*. La función verifica que todos los barcos de la lista pasada como argumento estén ubicados en el tablero.

```
def chequear_barcos(tablero, barcos):
    n = len(tablero)
    m = len(tablero[0])

    for i in range(n):
        for j in range(m):

            if len(barcos) == 0:
                break

            if tablero[i][j] == 1:
                largo_barco = 1
                tablero[i][j] = 0
                # Barco horizontal
                if j < m-1 and tablero[i][j+1] == 1:
                    while j < m-1:
                        if tablero[i][j+1] == 1:
                            largo_barco += 1
                            tablero[i][j+1] = 0
                            j += 1
                        else:
                            break

                # Barco vertical
                elif i < n-1 and tablero[i+1][j] == 1:
                    while i < n-1:
                        if tablero[i+1][j] == 1:
                            largo_barco += 1
                            tablero[i+1][j] = 0
                            i += 1
                        else:
                            break

                if barcos.count(largo_barco) > 0:
                    barcos.remove(largo_barco)
                else:
                    # No existe un barco con esa longitud
                    return False

    # Quedaron barcos sin asignar al tablero
    if len(barcos) != 0:
        return False

    return True
```

- Primero el algoritmo itera sobre todas las celdas del tablero ($O(n \times m)$.
- Por cada celda, si encuentra un 1 almacenado quiere decir que hay un barco ubicado. De lo contrario sigue con la siguiente celda.
- Si encuentra un barco, se analiza si es un barco ubicado de forma horizontal o vertical, verificando si en la celda hacia la derecha o hacia abajo hay un 1 almacenado.
- En cualquier caso, itera sobre la direccion horizontal (o vertical) hasta no encontrar más 1s. Cuando finaliza, analiza si existe un barco con ese tamaño y lo elimina de la lista de barcos.

- En el peor caso (en términos de complejidad) todos los barcos estarán ubicados, por lo que se iterarán la suma del tamaño de todos los barcos, dando una complejidad de $O(sum(barcos))$

Conclusión La complejidad corregida del validador propuesto es $O(n \times m \times sum(barcos))$, lo que es polinomial respecto de los valores de entrada, por lo que podemos afirmar que La Batalla Naval Individual está en NP.

3.8.4. Mejora en medición de complejidad empírica

Se nos pidió realizar mediciones con una mayor cantidad de sets de datos y una mayor variabilidad. Para eso, primero extenderemos las mediciones ya realizadas para tener una mayor cantidad de muestras, y además, haremos un análisis de cómo las variables afectan a los tiempos de ejecución.

Aclaraciones A contrario de la primera entrega, esta vez la generación de demandas y el tamaño de los barcos no se hará de forma aleatoria, ya que producía comportamientos erráticos en el algoritmo. Para un tablero de tamaño $(i \times i)$, las filas y columnas tendrán una demanda de i y habrá que ubicar $\frac{i}{2}$ barcos de tamaño $\frac{i}{2}$.

Mediciones Esta vez, se generarán 50 sets de datos que irán incrementando iterativamente el tamaño del tablero y la cantidad de barcos para medir con una mayor precisión.

A continuación se muestra el gráfico de las mediciones realizadas, relacionando el número de set (el set i tiene un tablero de $(i \times i)$ con $\frac{i}{2}$ barcos de tamaño $\frac{i}{2}$ y demandas con valor i) con el tiempo de ejecución:

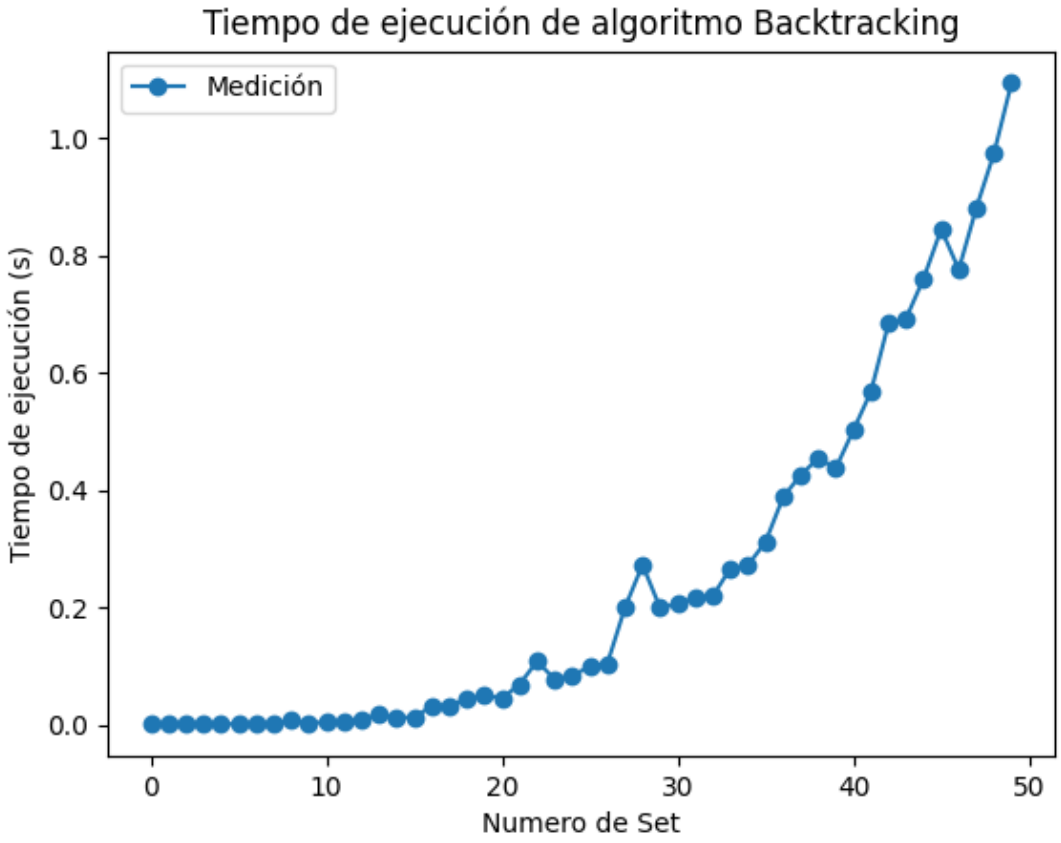

Figura 11: Mediciones temporales para el algoritmo de Backtracking

Como vemos, el algoritmo tiene una clara tendencia exponencial cuando se van incrementando el valor de sus variables.

Análisis de la variable: número de barcos A continuación, se realizarán mediciones de tiempo con la siguiente configuración:

- Se dejará el tamaño del tablero fijo en 20x20.
- Tanto las demandas de filas como de columnas tendrán un valor de 10.
- El tamaño de los barcos tendrá un valor fijo de 2.
- Se irá incrementando iterativamente el número de barcos, empezando desde 2 barcos hasta 30 barcos.

Con esta configuración, buscamos analizar cómo afecta la variable cantidad de barcos a los tiempos de ejecución del algoritmo de backtracking.

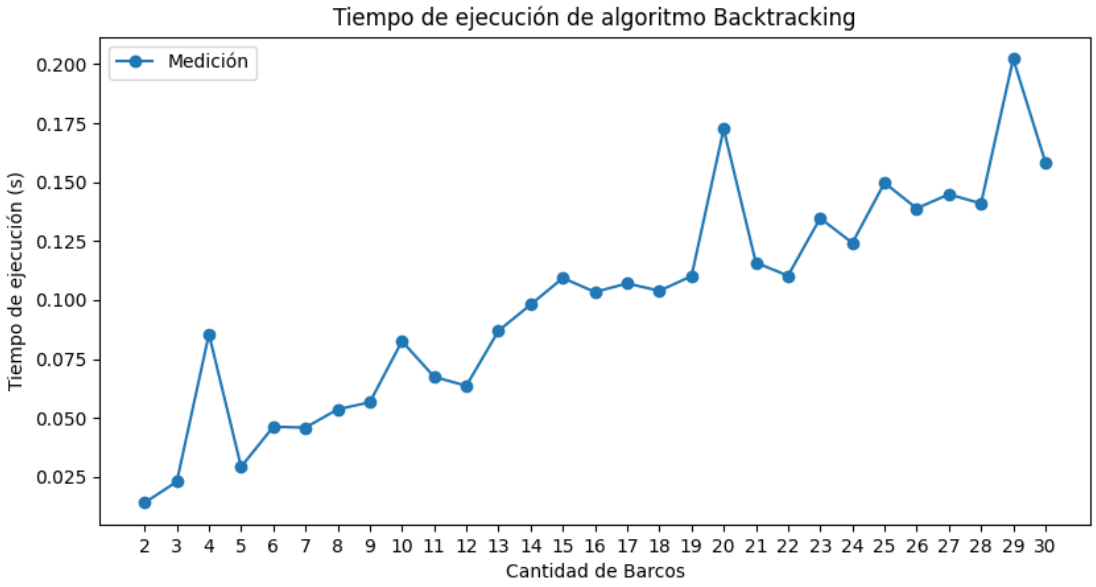

Figura 12: Mediciones temporales para el algoritmo de Backtracking incrementando el numero de barcos

Como vemos, aunque se observan algunas fluctuaciones en los valores individuales, la tendencia general sugiere un aumento lineal del tiempo de ejecución a medida que la cantidad de barcos aumenta.

Análisis de la variable: tamaño del tablero A continuación, se realizarán mediciones de tiempo con la siguiente configuración:

- Se irá incrementando iterativamente el tamaño del tablero desde 10x10 hasta 30x30.
- Tanto las demandas de filas como de columnas tendrán un valor de 10.
- El tamaño de los barcos tendrá un valor fijo de 3.
- Se tendrá un número fijo de 7 barcos.

Con esta configuración, buscamos analizar cómo afecta la variable tamaño del tablero a los tiempos de ejecución del algoritmo de backtracking.

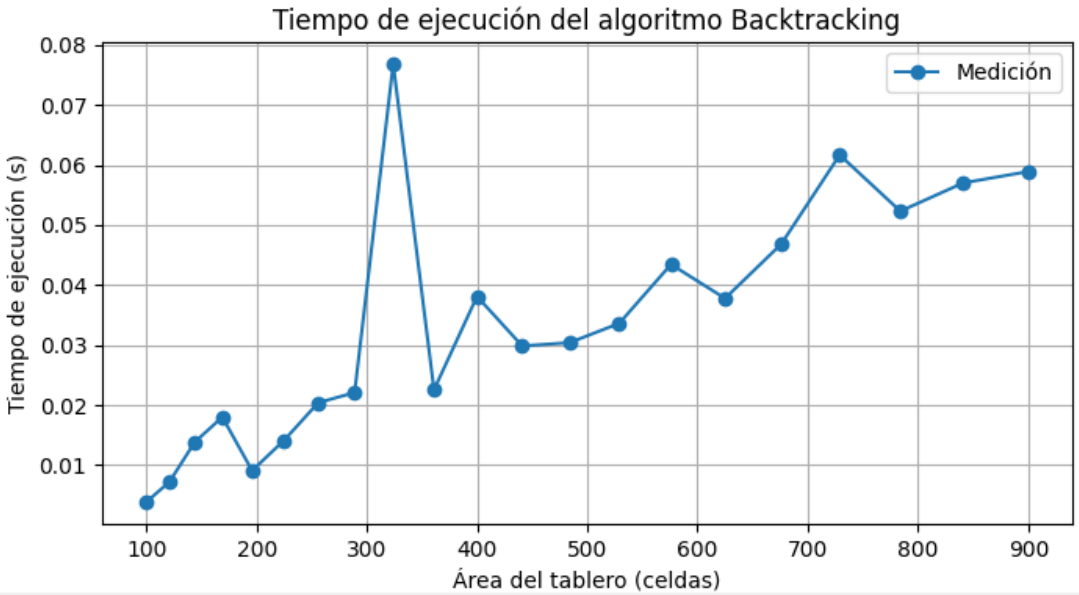

Figura 13: Mediciones temporales para el algoritmo de Backtracking incrementando el tamaño del tablero

Nuevamente, a pesar de algunas fluctuaciones vemos que la tendencia general es que el tiempo de ejecución del algoritmo incrementa linealmente a medida que incrementamos el tamaño del tablero.

Análisis de la variable: tamaño de los barcos

A continuación, se realizarán mediciones de tiempo con la siguiente configuración:

- Se dejará el tamaño del tablero fijo en 25x25.
- Tanto las demandas de filas como de columnas tendrán un valor de 20.
- El tamaño de los barcos irá incrementando iterativamente desde tamaño 1 hasta tamaño 15.
- Se tendrá un número fijo de 10 barcos a ubicar.

Con esta configuración, buscamos analizar cómo afecta la variable tamaño de los barcos a los tiempos de ejecución del algoritmo de backtracking.

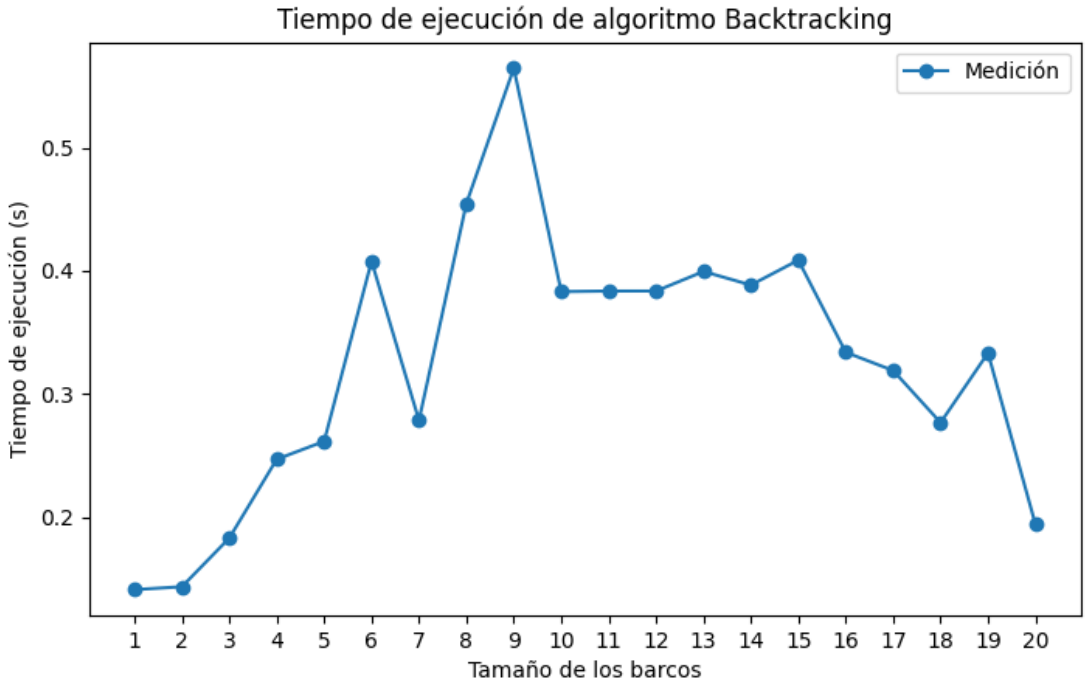

Figura 14: Mediciones temporales para el algoritmo de Backtracking incrementando el tamaño de los barcos

Esta vez el análisis no es tan simple. Podemos observar como para barcos de pequeño tamaño, el algoritmo se comporta con bastante rapidez, pero a medida que crece el tamaño de los barcos parecería crecer el tiempo exponencialmente. Sin embargo, cuando los barcos son muy grandes, nuevamente el algoritmo entrega la solución con rapidez. Esto nos hace pensar que el tamaño de los barcos influye significativamente al tiempo de ejecución del algoritmo.

3.8.5. Corrección implementación de Algoritmo de Aproximación

Se nos remarcó que la implementación algoritmo de aproximación no funcionaba correctamente. Esto era debido a que si no se podía ubicar ningún barco en la fila/columna de mayor demanda, entonces el algoritmo cortaba su ejecución, perdiéndose de posibilidades válidas.

En este caso, el algoritmo si no encuentra cómo ubicar algun barco en la fila/columna de mayor demanda, no corta la ejecución, sino que sigue con la siguiente fila/columna de mayor demanda.

El código corregido se muestra a continuación:

Algoritmo propuesto

```
1     def batalla_naval_individual_aprox(tablero, demanda_filas, demanda_columnas,
      barcos):
2         indice_barco = 1
3         n = len(tablero)
4         m = len(tablero[0])
5
6         while queda_demanda_por_cumplir(demanda_filas, demanda_columnas) and barcos
      :
7             flag_marcado = False
8
9             # Obtenemos todos los indices de las demandas junto con su orientacion
      (fila/columna)
10            demandas = [(i, "fila", demanda_filas[i]) for i in range(n) if
      demanda_filas[i] > 0] + \
```

```
                        [(j, "columna", demanda_columnas[j]) for j in range(m) if demanda_columnas[j] > 0]

            # Ordenar por demanda de mayor a menor
            demandas.sort(key=lambda x: x[2], reverse=True)

            for indice_max_demanda, tipo, _ in demandas:
                for barco in barcos:
                    posicion = posicion_valida(tablero, barco, n, m, demanda_filas, demanda_columnas, indice_max_demanda, tipo)

                    if posicion is not None and not supera_demanda_permitida(tablero, posicion, barco, demanda_filas, demanda_columnas):
                        if tipo == "fila" and demanda_filas[indice_max_demanda] >= barco:
                            marcar_barco_y_actualizar_demandas(tablero, barco, posicion, demanda_filas, demanda_columnas)
                            barcos.remove(barco)
                            indice_barco += 1
                            flag_marcado = True
                            break
                        elif tipo == "columna" and demanda_columnas[indice_max_demanda] >= barco:
                            marcar_barco_y_actualizar_demandas(tablero, barco, posicion, demanda_filas, demanda_columnas)
                            barcos.remove(barco)
                            indice_barco += 1
                            flag_marcado = True
                            break

                if flag_marcado:
                    break  # Si pudimos colocar un barco, volvemos a calcular demandas

            # Si no pudimos colocar ningun barco en ninguna fila/columna, terminamos
            if not flag_marcado:
                break

        return tablero, max(0, sum(demanda_filas) + sum(demanda_columnas))
```

Análisis de la complejidad A continuación analizaremos los fragmentos del algoritmo planteado y su complejidad, para poder analizar la complejidad de la totalidad del algoritmo de aproximación.

Antes de llamar al algoritmo, se realizan algunas operaciones como:

- Construir el tablero vacio: $O(n \times m)$
- Ordenar los barcos de mayor a menor: $O(klog(k))$, siendo k la cantidad de barcos.

La función principal del algoritmo itera mientras haya demandas o barcos $O(k)$, y en cada iteracion se realizan las siguientes operaciones:

Siendo N la suma entre las filas n y columnas m del tablero y B el máximo tamaño de un barco:

- Obtener un arreglo de demandas, con una complejidad de $O(N)$
- Ordenar el arreglo de demandas, con una complejidad de $O(N \log(N))$
- En el peor caso, siempre la demanda de menor valor es la única que puede ubicar un barco, por lo que itera todo el arreglo de demandas en $O(N)$:
    - También en el peor caso, itera toda la lista de barcos en $O(k)$ y por cada barco:
        - Analizar si hay una posición válida $O(max(n, m) * B)$
        - Marcar el barco en el tablero y actualizar las demandas $O(B)$

Desarrollando la complejidad nos queda $O(k * [N + N * \log(N) + N * k * (max(n, m) * B + B)])$. Podemos ver que el tercer término es el predominante por lo que decimos que la complejdiad de la lógica principal del algoritmo es $O(k^2 * N * (max(n, m) + B))$

Por lo tanto, la complejidad total del algoritmo de aproximación, teniendo en cuenta tanto la preparación de la solución como el bucle principal es:

$O(n * m + k * log(k) + k^2 * N * (max(n, m) + B))$

Comparándolo con las soluciones por Backtracking o Programación Lineal Entera, que ambos tienen complejidad exponencial, parece reducir significativamente los tiempos de ejecución, ya que como peor caso no parece superar la complejidad cuadrática.

Mediciones Con el algoritmo corregido, mostramos a continuación las mediciones obtenidas utilizando el mismo set de datos que para el algoritmo de backtracking.

Recordamos que el set i tiene un tablero de $(i \times i)$ con $\frac{i}{2}$ barcos de tamaño $\frac{i}{2}$ y demandas con valor i.

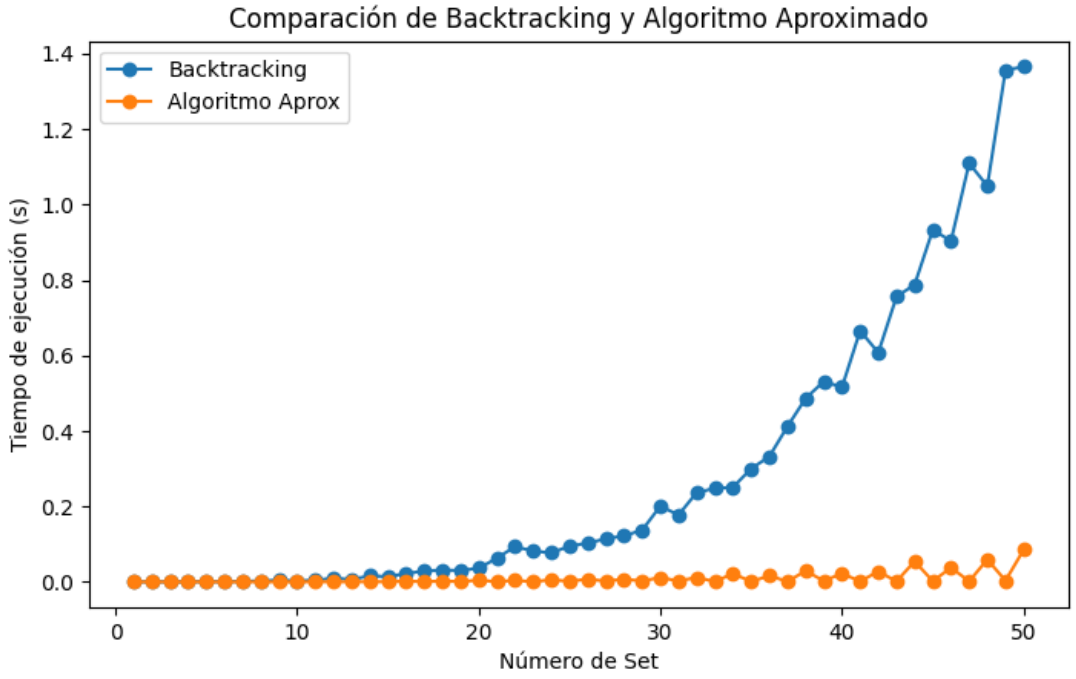

Figura 15: Gráfico comparativo entre el algoritmo de Backtracking y el algoritmo de Aproximación

Ahora si vemos como claramente el algoritmo de aproximación es significativamente más rápido que el de backtracking, que tiene complejidad exponencial.

Análisis de la aproximación En la primera entrega se midió mal la cota $r(A)$, dando resultados erróneos. A continuación se medirá cuán buena es la aproximación del algoritmo planteado, comparando empíricamente los resultados arrojados contra la solución óptima.

Recordamos:

- I es una instancia cualquiera del problema de La Batalla Naval.
- $A(I)$ es la solución entregada por el algoritmo de aproximación de la instancia I.
- $z(I)$ es la solución óptima (medida con el algoritmo de Backtracking) para la instancia I.

- Se define $\frac{A(I)}{z(I)} \leq r(A)$ como una cota para demostrar cuán buena es la aproximación.

Calculamos la cota para los sets de datos brindados por la cátedra, se toma como solución a la **demanda incumplida** arrojada por el algoritmo:

Caso de Prueba (I)	$z(I)$	$A(I)$	$r(A)$
3_3_2.txt	7	7	1
5_5_6.txt	6	6	1
8_7_10.txt	27	27	1
10_3_3.txt	8	8	1
10_10_10.txt	0	2	-
12_12_21.txt	12	18	1.5
15_10_15.txt	27	27	1
20_20_20.txt	16	34	2.125
20_25_30.txt	75	95	1.26
30_25_25.txt	158	190	1.202

Cuadro 6: Tabla comparativa entre solución óptima y solución aproximada con casos de prueba.

Podemos ver cómo generalmente, la cota de aproximación para los casos de prueba de tamaño pequeño es muy baja, al punto que el resultado es idéntico. Sin embargo, para casos de prueba más grandes, la cota de aproximación tiende a ser mayor. Para los casos de prueba con tablero mayor a 10x10, el promedio de la cota de aproximación es de 1.4174, siendo esta una aproximación bastante buena. Si miramos más detalladamente, vemos como en casos particulares, la aproximación puede dar un muy mal resultado, como vemos en el caso de prueba 20_20_20.*txt*, donde la cota dió 2.125, es decir, que el resultado obtenido es hasta dos veces peor que el óptimo. Si tomamos el peor caso como la cota obtenida, podríamos concluir que la aproximación es mala, sin embargo, entendemos que este es un caso particular y concluímos que generalmente, la aproximación da buenos resultados.